Scheme: struttura del programma e campo di azione

Nel capitolo introduttivo, sono state elencate le strutture elementari per il controllo e il raggruppamento delle istruzioni (espressioni) di Scheme. In questo capitolo, si vuole mostrare in che modo possano essere definite delle funzioni, o comunque dei raggruppamenti di istruzioni all'interno dei quali si possa individuare un campo di azione locale per le variabili che vi vengono dichiarate.

Le funzioni del linguaggio Scheme prevedono il passaggio di parametri solo **per valore**; questo significa che gli argomenti di una funzione vengono valutati prima di tutto. Al posto del passaggio dei parametri per riferimento, Scheme consente l'indicazione di espressioni costanti, concetto a cui si è accennato nel capitolo precedente.

307.1 Definizione e campo di azione

La definizione e inizializzazione di un oggetto avviene normalmente attraverso la funzione '**`define`**'. Questa può servire per dichiarare una variabile normale, o anche per dichiarare una funzione.

```
(define nome_variabile espressione_di_inizializzazione )
```

Quello che si vede sopra è appunto lo schema sintattico per la dichiarazione e inizializzazione di una variabile, cosa che è stata vista più volte nel capitolo precedentemente. Sotto, si vede lo schema sintattico per la dichiarazione di una funzione:

```
(define (nome_funzione elenco_parametri_formali )
    corpo
)
```

In questo caso, i parametri formali sono una serie di nomi che rappresentano i parametri della funzione che viene dichiarata, mentre il corpo è costituito da una serie di espressioni, che rappresentano il contenuto della funzione che si dichiara. Il valore che viene restituito dall'ultima espressione che viene eseguita all'interno della funzione, è ciò che restituisce la funzione stessa. L'esempio seguente, serve a definire la funzione '**`moltiplica`**' con due parametri, '**`x`**' e '**`y`**', che restituisce il prodotto dei suoi due argomenti:

```
(define (moltiplica x y)
    ; il corpo di questa funzione è molto breve
    (* x y)
)
```

Per chiamare questa funzione, basta semplicemente un'istruzione come quella seguente:

```
(moltiplica 10 11)              ===> 110
```

Le dichiarazioni di questo tipo, cioè di variabili e di funzioni, possono avvenire solo nella parte più esterna di un programma Scheme, oppure all'interno della dichiarazione di altre funzioni e delle altre strutture descritte in questo capitolo, ma in tal caso devono apparire all'inizio del «corpo» delle espressioni che queste strutture contengono. Si osservi l'esempio seguente, in cui viene dichiarata una funzione e al suo interno si dichiarano altre variabili locali:

```
(define (moltiplica x y)
    ; dichiara le variabili locali
    (define z 0)

    ; definisce un ciclo enumerativo, per il calcolo del prodotto
    ; attraverso la somma, in cui viene dichiarata implicitamente
```

```
    ; la variabile «i».
    (do ((i 1 (+ i 1)))
        ; condizione di uscita
        ((> i y))
        ; istruzioni del ciclo
        (set! z (+ z x))
    )
    ; al termine restituisce il valore contenuto nella variabile «z»
    z
)
```

Dovrebbe essere intuitivo, quindi, che il campo di azione delle variabili dichiarate all'interno di una funzione '**`define`**' è limitato alla funzione stessa. La stessa cosa varrebbe per le funzioni, dichiarate all'interno di un ambiente del genere. Si osservi l'esempio seguente, in cui si calcola il prodotto tra due numeri, a partire dalla somma di questi, ma dove la somma si ottiene da un'altra funzione, locale, che a sua volta la calcola con incrementi di una sola unità alla volta.

```
(define (moltiplica x y)
    ; dichiara la funzione «somma», locale nell'ambito della
    ; funzione «moltiplica»
    (define (somma x y)
        ; dichiara una variabile locale per la funzione «somma»,
        ; che comunque non serve a nulla :-)
        (define z 2000)
        ; definisce un ciclo enumerativo, per il calcolo della
        ; somma, sommando un'unità alla volta
        (do ()
            ; condizione di uscita
            ((<= y 0))
            ; istruzioni del ciclo
            (set! x (+ x 1))
            ; decrementa «y»
            (set! y (- y 1))
        )
        ; al termine restituisce il valore contenuto nella variabile «x»
        x
    ; fine della funzione locale «somma»
    )
    ; dichiara le variabili locali della funzione «moltiplica»
    (define z 0)
    ; definisce un ciclo enumerativo, per il calcolo del prodotto
    ; attraverso la somma, in cui viene dichiarata implicitamente
    ; la variabile «i».
    (do ((i 1 (+ i 1)))
        ; condizione di uscita
        ((> i y))
        ; istruzioni del ciclo
        (set! z (somma z x))
    )
    ; al termine restituisce il valore contenuto nella variabile «z»
    z
)
```

Questo esempio è solo un pretesto per mostrare che le variabili locali '**`x`**', '**`y`**' e '**`z`**', della funzione '**`somma`**' hanno effetto solo nell'ambito di questa funzione; inoltre, la funzione '**`somma`**' e le variabili locali '**`x`**', '**`y`**' e '**`z`**', della funzione '**`moltiplica`**', hanno effetto solo nell'ambito della funzione '**`moltiplica`**' stessa.

307.1.1 Ridefinizione

Nel capitolo introduttivo si è accennato al fatto che la ridefinizione di una variabile, o di una funzione, implica una nuova allocazione di memoria, senza liberare quella utilizzata precedentemente. Questo implica che i riferimenti fatti in precedenza a quell'oggetto, continuano a utilizzare in pratica la vecchia allocazione. Si osservi l'esempio seguente:

```
(define x 20)
x                       ===> 20
(define y (* 2 x))
y                       ===> 40
(define x 100)
x                       ===> 100
y                       ===> 40
```

Quanto mostrato con questo esempio, non ha nulla di eccezionale, rispetto ai linguaggi di programmazione tradizionali. Tuttavia, potrebbe risultare strano da un punto di vista strettamente matematico. Se invece lo scopo fosse quello di definire un sistema di equazioni, '**y**' dovrebbe essere trasformato in una funzione, come nell'esempio seguente:

```
(define x 20)
x                       ===> 20
(define (f y) (* 2 x))
(f)                     ===> 40
(define x 100)
x                       ===> 100
(f)                     ===> 200
```

Qualunque oggetto con un identificatore può essere ridefinito. Si osservi l'esempio seguente, in cui si imbrogliano le carte e si fa in modo che l'identificatore '*****' corrisponda a una funzione che esegue la somma, mentre prima valeva per una moltiplicazione:

```
(define (* x y) (+ x y)
(* 3 5)                 ===> 8
```

Si ricorda che per modificare il contenuto di una variabile allocata, senza allocare un'altra area di memoria, si utilizza generalmente la funzione '**set!**'.

307.2 Definizione «lambda»

Scheme tratta gli identificatori delle funzioni (i loro nomi), nello stesso modo di quelli delle variabili. In altri termini, le funzioni sono variabili che contengono un riferimento a un blocco di codice. È possibile dichiarare una funzione attraverso la funzione '**lambda**', che restituisce la funzione stessa. In questo modo, una funzione può essere dichiarata anche attraverso l'assegnamento di una variabile, che poi diventa una funzione a tutti gli effetti.

Prima di vedere come si usa la dichiarazione di una funzione attraverso la funzione '**lambda**', è bene ribadire che, attraverso questo meccanismo, è possibile dichiarare una funzione in tutte quelle situazioni in cui è possibile inizializzare o assegnare una variabile.

```
(lambda (elenco_parametri_formali )
     corpo
)
```

Come si vede dal modello sintattico, la funzione '**lambda**' è relativamente semplice: il primo argomento è un blocco contenente l'elenco dei nomi (locali) dei parametri formali; gli argomenti successivi sono le espressioni che costituiscono il corpo della funzione. Non si dichiara il nome della funzione, dal momento che '**lambda**' restituisce la funzione stessa, che verrà identificata (ammesso che lo si voglia fare) dalla variabile a cui questa viene assegnata.

All'inizio del «corpo» delle espressioni che descrivono il contenuto della funzione che si dichiara, si possono inserire delle dichiarazioni ulteriori attraverso la funzione '**`define`**'.

Sotto vengono proposti alcuni esempi che dovrebbero lasciare intendere in quante situazioni si può utilizzare una dichiarazione di funzione attraverso '**`lambda`**'.

```
; dichiara la variabile «f» e la inizializza temporaneamente al valore zero
(define f 0)
; assegna a «f» una funzione che esegue la somma dei suoi due argomenti
(set! f
    (lambda (x y)
        (+ x y)
    )
)
; calcola la somma tra 4 e 5, restituendo 9
(f 4 5)
```

L'esempio che appare sopra mostra in che modo si possa dichiarare una funzione in qualunque situazione in cui si può assegnare un valore a una variabile.

```
; dichiara direttamente la funzione «f»
(define f
    ; inizializza «f» con una funzione che esegue la somma
    ; dei suoi due argomenti
    (lambda (x y)
        ; corpo della dichiarazione della funzione
        (+ x y)
    )
)
; calcola la somma tra 4 e 5, restituendo 9
(f 4 5)
```

In questo caso, l'assegnamento della funzione alla variabile '**`f`**' è avvenuto contestualmente alla dichiarazione della variabile stessa.

```
(define moltiplica
    (lambda (x y)
        ; dichiara le variabili locali
        (define z 0)
        ; definisce un ciclo enumerativo, per il calcolo del prodotto
        ; attraverso la somma, in cui viene dichiarata implicitamente
        ; la variabile «i».
        (do ((i 1 (+ i 1)))
            ; condizione di uscita
            ((> i y))
            ; istruzioni del ciclo
            (set! z (+ z x))
        )
        ; al termine restituisce il valore contenuto nella variabile «z»
        z
    )
)
```

Questo esempio, mostra in che modo possano avvenire delle dichiarazioni locali nel corpo di una dichiarazione '**`lambda`**'.

L'esempio successivo è un po' un estremo, nel senso che viene mostrata la dichiarazione di una funzione «anonima», che viene usata immediatamente per calcolare il prodotto tra tre e quattro. Successivamente al suo utilizzo istantaneo, non c'è modo di riutilizzare tale funzione.

```
(
    ; dichiarazione della funzione anonima
    (lambda (x y)
        ; dichiara le variabili locali
        (define z 0)
        ; definisce un ciclo enumerativo, per il calcolo del prodotto
```

```
        ; attraverso la somma, in cui viene dichiarata implicitamente
        ; la variabile «i».
        (do ((i 1 (+ i 1)))
            ; condizione di uscita
            ((> i y))
            ; istruzioni del ciclo
            (set! z (+ z x))
        )
        ; al termine restituisce il valore contenuto nella variabile «z»
        z
    )
    ; indicazione del primo argomento
    3
    ; indicazione del secondo argomento
    4
)
```

307.3 Ricorsione

Si intuisce la possibilità di Scheme di scrivere funzioni ricorsive. Non dovrebbe essere difficile arrivare a questo risultato senza spiegazioni particolari. L'esempio seguente mostra il calcolo del fattoriale attraverso una funzione ricorsiva.

```
(define (fattoriale n)
    (if (= n 0)
        ; then
        1
        ; else
        (* n (fattoriale (- n 1)))
    )
)
```

Si intuisce che una funzione senza nome, come nel caso di quella dichiarata con '**`lambda`**', senza assegnarla a una variabile, non può essere resa ricorsiva, a meno di definire una sotto-funzione ricorsiva al suo interno. L'esempio seguente è una variante di quello precedente, in cui viene utilizzata una dichiarazione '**`lambda`**'.

```
(define fattoriale
    (lambda (n)
        (if (= n 0)
            ; then
            1
            ; else
            (* n (fattoriale (- n 1)))
        )
    )
)
```

307.4 let, let* e letrec

Le funzioni '**`let`**', '**`let*`**' e '**`letrec`**', hanno lo scopo di circoscrivere un ambiente, all'interno del quale può essere inserita una serie indefinita di espressioni (istruzioni), prima delle quali vengono dichiarate delle variabili il cui campo di azione è locale rispetto a quell'ambito.

```
(let ((variabile inizializzazione)...)
    corpo
)
```

```
(let* ((variabile inizializzazione)...)
    corpo
)
```

```
(letrec ((variabile inizializzazione)...)
    corpo
)
```

In tutti e tre le forme, le variabili vengono inizializzate e quindi si passa alla valutazione delle espressioni successive (le istruzioni). Alla fine, la funzione restituisce il valore dell'ultima espressione a essere stata eseguita al suo interno.

Nel caso di '**`let`**', le variabili vengono dichiarate e inizializzate senza un ordine preciso, ma semplicemente prima di passare alla valutazione delle espressioni successive:

```
(let ((x 1) (y 2))
    (+ x y)
)                       ===> 3
```

L'esempio non ha un grande significato da un punto di vista pratico, ma si limita a mostrare intuitivamente come si comporta la funzione '**`let`**'. In questo caso, vengono dichiarate le variabili locali '**`x`**' e '**`y`**', inizializzandole rispettivamente a uno e due, infine viene calcolata semplicemente la somma tra le due variabili, cosa che restituisce il valore tre.

Nel caso di '**`let*`**', le variabili vengono dichiarate e inizializzate nell'ordine in cui sono (da sinistra a destra); pertanto, ogni inizializzazione può fare riferimento alle variabili dichiarate precedentemente nella stessa sequenza:

```
(let* ((x 1) (y (+ x 1)))
    (+ x y)
)                       ===> 3
```

L'esempio mostra che la variabile locale '**`y`**' viene inizializzata partendo dal valore della variabile locale '**`x`**', incrementando il valore di questa di un'unità.

La funzione '**`letrec`**' è più sofisticata; il nome sta per *let recursive*. È un po' difficile spiegare il senso di questa; si tenta almeno di mostrare la cosa in modo intuitivo.

Nello stesso modo in cui si può dichiarare una variabile, si può dichiarare una funzione. In questo senso, tali dichiarazioni possono anche essere ricorsive all'interno di una funzione '**`letrec`**'. Viene mostrato un esempio tratto da *R*^{5}*RS*:

```
(letrec
    ; dichiara le «variabili», che in realtà sono funzioni (predicati)
    (
        ; dichiara la funzione «pari?»
        (pari?
            (lambda (n)
                (if (zero? n)
                    ; il numero è pari
                    #t
                    ; altrimenti si prova a vedere se è dispari
                    (dispari? (- n 1))
                )
            )
        )
        ; dichiara la funzione «dispari?»
        (dispari?
            (lambda (n)
                (if (zero? n)
                    ; il numero è dispari
                    #f
                    ; altrimenti si prova a vedere se è pari
                    (pari? (- n 1))
                )
            )
        )
    )
    ; fine della dichiarazione delle variabili

    ; verifica che il numero 88 è pari, chiamando la funzione
    ; «pari?» dichiarata all'inizio
    (pari? 88)
```

```
    ; la chiamata restituisce il valore #t e, di conseguenza,
    ; è questo il valore restituiti da tutto
)
```

Le variabili '**pari?**' e '**dispari?**' vengono inizializzate assegnando loro una funzione dichiarata con '**lambda**' e il loro scopo è quello di verificare che l'argomento sia rispettivamente un numero pari o dispari.

```
(pari? 2)               ===> #t
(dispari? 2)            ===> #f
```

Tali variabili e di conseguenza queste funzioni, hanno effetto solo nell'ambito della dichiarazione '**letrec**', al termine della quale diventano semplicemente irraggiungibili. Il principio di funzionamento di queste funzioni, sta nel fatto che lo zero sia pari, di conseguenza:

```
(pari? 0)               ===> #t
(dispari? 0)            ===> #f
```

Per tutti i numeri superiori, invece, è sufficiente verificare in modo ricorsivo di che tipo è il valore n - 1. Per la precisione, se si sta verificando il fatto che un numero sia pari, se questo è superiore a zero, si può verificare che quel numero, meno uno, sia dispari, continuando così, di seguito.

Queste tre strutture sono importanti soprattutto perché consentono di inserire delle dichiarazioni di variabili o di funzioni, oltre al fatto che così circoscrivono un ambito locale per queste. Come si è visto, queste dichiarazioni possono essere fatte anche prima (anche con '**let**' e '**let***'), tenendo conto dell'ordine di valutazione che ognuna di queste strutture garantisce.

```
(let ((x 1) (y 2))
    (define messaggio "sto calcolando la somma...")
    (display messaggio)
    (newline)
    (+ x y)
)                       ===> 3
```

L'esempio che si vede sopra, è solo un'estensione di quanto già visto sopra, allo scopo di mostrare la possibilità di utilizzare la funzione '**define**' all'inizio del corpo di espressioni che contiene. L'esempio successivo è una variante ulteriore, in cui il messaggio viene dichiarato tra le variabili iniziali di '**let**'.

```
(let ((x 1) (y 2) (messaggio "sto calcolando la somma..."))
    (display messaggio)
    (newline)
    (+ x y)
)                       ===> 3
```

Scheme: liste e vettori

Scheme dispone di due strutture di dati particolari: liste e vettori. Le liste sono una sequenza di elementi a cui si accede con una certa difficoltà, senza la possibilità di utilizzare un indice, mentre i vettori sono l'equivalente degli array degli altri linguaggi.

308.1 Liste e coppie

La lista è la struttura di dati fondamentale di Scheme. In questo linguaggio, le stesse istruzioni (le chiamate delle funzioni) sono espresse in forma di lista:

```
(elemento ...)
```

La lista è un elenco di elementi ordinati. Gli elementi di una lista possono essere oggetti di qualunque tipo, comprese altre liste. Ci sono molte situazioni in cui i parametri di una funzione di Scheme sono delle liste; per esempio la dichiarazione di una funzione, attraverso '**`define`**':

```
(define (nome_funzione elenco_parametri_formali )
    corpo
)
```

Come si vede, il primo parametro della funzione '**`define`**' è una lista, in cui il primo elemento è il nome della funzione che si crea, mentre gli elementi successivi sono la descrizione dei parametri formali.

Le liste vuote, sono rappresentate da una coppia di parentesi aperta e chiusa, '**`()`**', rappresentando degli oggetti speciali nella filosofia di Scheme.

Tabella 308.1. Elenco di alcune funzioni specifiche per la gestione delle stringhe.

Funzione	Descrizione
(list? *oggetto*)	*Vero* se l'oggetto è una lista.
(pair? *oggetto*)	*Vero* se l'oggetto è una coppia (una lista non vuota).
(null? *lista*)	*Vero* se la lista è vuota.
(length *lista*)	Restituisce il numero di elementi della lista.
(car *lista*)	Restituisce il primo elemento di una lista.
(cdr *lista*)	Restituisce una lista da cui è stato tolto il primo elemento.
(cadr *lista*)	Equivale a (car (cdr *lista*))
(cddr *lista*)	Equivale a (cdr (cdr *lista*))
(caadr *lista*)	Equivale a (car (car (cdr *lista*)))
(caddr *lista*)	Equivale a (car (cdr (cdr *lista*)))
(cons *elemento lista*)	Restituisce una lista in cui inserisce al primo posto l'elemento indicato.
(list *elemento ...*)	Restituisce una lista composta dagli elementi indicati.
(append *lista lista*)	Restituisce una lista composta dagli elementi delle due liste indicate.
(reverse *lista*)	Restituisce una lista con gli elementi in ordine inverso.
(set-car! *lista oggetto*)	Memorizza nella prima posizione della lista l'oggetto indicato.
(set-cdr! *lista oggetto*)	Memorizza nella parte successiva al primo elemento l'oggetto indicato.
(list-tail *lista k*)	Restituisce una lista in cui mancano i primi *k* elementi.
(list-ref *lista k*)	Restituisce l'elemento (k + 1)-esimo della lista.
(vector->list *vettore*)	Converte il vettore in lista.
(list->vector *list*)	Converte la lista in vettore.

308.1.1 Dichiarazione di una lista

La dichiarazione di una lista avviene nello stesso modo in cui si dichiara una variabile normale:

```
(define variabile lista_costante)
```

Tuttavia, occorre tenere presente che una lista può essere interpretata come la chiamata di una funzione e come tale verrebbe intesa in questa situazione. Per evitare che ciò avvenga, la si indica attraverso un'espressione costante, cioè la si fa precedere da un apostrofo, o la si inserisce in una funzione '**quote**'. L'esempio seguente dichiara la lista '**lis**' composta dall'elenco dei numeri interi da uno a sei:

```
(define lis '(1 2 3 4 5 6))
```

In questo caso, se la lista non venisse indicata con l'apostrofo, si otterrebbe la valutazione della lista stessa, prima dell'inizializzazione della variabile '**lis**', provocando un errore, dal momento che l'oggetto '**1**' (uno) non esiste.

308.1.2 Caratteristiche esteriori di una lista

Le caratteristiche esteriori di una lista sono semplicemente la lunghezza, espressa in numero di elementi, e il fatto che contengano o meno qualcosa. Per verificare queste caratteristiche sono disponibili due funzioni, '**null?**' e '**length**', che richiedono come argomento una lista. Si osservino gli esempi seguenti.

```
; dichiara la lista «lis»
(define lis (1 2 3 4 5 6)

; verifica se la lista «lis» è vuota
(null? lis)                                    ===> #f

; calcola la lunghezza della lista
(length lis)                                   ===> 6
```

Se fosse stata fornita la lista in modo letterale, senza la variabile '**lis**', la stessa cosa avrebbe dovuto essere scritta nel modo seguente:

```
; verifica se la lista è vuota
(null? '(1 2 3 4 5 6))                         ===> #f

; calcola la lunghezza della lista
(length '(1 2 3 4 5 6))                        ===> 6
```

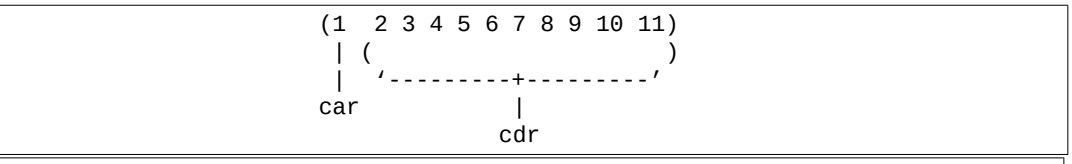
308.1.3 Operazioni fondamentali con le liste

L'accesso agli elementi singoli di una lista è un'impresa piuttosto complessa che si attua fondamentalmente con le funzioni '**car**' e '**cdr**'. A queste due si affianca anche '**cons**', il cui scopo è quello di «costruire» una lista.

Per comprendere il senso di queste funzioni, occorre tenere presente che per Scheme una lista è una *coppia* composta dal primo elemento, ovvero l'elemento '**car**', e dalla parte restante, ovvero la parte '**cdr**'.

> Per la precisione, una coppia è una lista, mentre la lista vuota non è una coppia. La lista contenente un solo elemento, è la composizione dell'unico elemento a disposizione e della lista vuota.

Figura 308.1. La parte «car» e la parte «cdr» che compongono le liste di Scheme.



`(car` *lista*`)`

`(cdr` *lista*`)`

Le due funzioni '**car**' e '**cdr**' hanno come argomento una lista, della quale restituiscono, rispettivamente, il primo elemento e la lista restante quando si elimina il primo elemento. Si osservino gli esempi seguenti.[1]

```
(car '(1 2 3 4 5 6))            ===> 1
(cdr '(1 2 3 4 5 6))            ===> (2 3 4 5 6)
```

Data l'idea che ha Scheme sulle liste, la funzione '**cons**' crea una lista a partire dalle sue parti '**car**' e '**cdr**':

`(cons` *elemento_car lista_cdr* `)`

In altri termini, '**cons**' aggiunge un elemento all'inizio della lista indicata come secondo argomento. Si osservi l'esempio.

```
(cons 0 '(1 2 3 4 5 6))         ===> (0 1 2 3 4 5 6)
```

Le tre funzioni '**car**', '**cdr**' e '**cons**' si completano a vicenda, in base alla relazione schematizzata dalla figura 308.2.

> Se viene fornita una lista come primo argomento della funzione '**car**', questa viene inserita come primo elemento della lista risultante.

```
(cons '(0 1 2) '(1 2 3 4 5 6))          ===> ((0 1 2) 1 2 3 4 5 6)
```

Figura 308.2. Relazione che lega le funzioni '**car**', '**cdr**' e '**cons**'. In particolare, «x» e «y» sono liste non vuote; «a» è un elemento ipotetico di una lista.

```
(cons (car x) (cdr x)) = x
(car (cons a y)) = a
(cdr (cons a y)) = y
```

Altri modi per creare una lista sono dati dalle funzioni '**list**' e '**append**'.

`(list` *elemento…*`)`

`(append` *lista lista*`)`

La funzione '**list**' restituisce una lista composta dai suoi argomenti (se non si vuole che questi siano valutati prima, occorre ricordare di usare l'apostrofo); la funzione '**append**' restituisce una lista composta dagli elementi delle due liste indicate come argomento (se le liste vengono fornite in modo letterale, occorre ricordare di usare l'apostrofo, per evitare che vengano valutate come funzioni).

```
(list 1 2 3 4 5 6)                      ===> (1 2 3 4 5 6)
(append '(1 2 3 4 5 6) '(7 8 9))        ===> (1 2 3 4 5 6 7 8 9)
```

Per verificare che un oggetto sia una lista, è disponibile il predicato '**list?**'. Si osservi l'esempio seguente, con il quale si intende ribadire il significato dell'apostrofo per evitare che una lista sia interpretata come funzione:

```
(define a (+ 1 2))
a                               ===> 3
```

[1] A questo punto si intende ormai chiarito il significato dell'apostrofo posto di fronte a una lista, quando questa non deve essere valutata, prima di essere fornita come argomento di una funzione.

```
(define b '(+ 1 2))
b                               ===> (+ 1 2)

(list? a)                       ===> #f
(list? b)                       ===> #t
```

308.1.4 Funzioni tipiche sulle liste

Dal momento che con le liste di Scheme non è disponibile un accesso diretto all'elemento *n*-esimo, se non attraverso la funzione di libreria '**`list-ref`**', è importante imparare a gestire le funzioni elementari già mostrate nella sezione precedente.

- Calcolo della lunghezza di una lista:

```
(define (lunghezza x)
    (if (null? x)
        ; se la lista è vuota, restituisce zero
        0
        ; altrimenti esegue una chiamata ricorsiva
        (+ 1 (lunghezza (cdr x)))
    )
)
```

- Ricerca dell'elemento *i*-esimo, dove il primo è il numero uno (si veda anche la funzione di libreria '**`list-ref`**', descritta più avanti in questa serie di esempi):

```
(define (i-esimo-elemento i x)
    ; «i» è l'indice, «x» è la lista
    (if (null? x)
        ; la lista è più corta di «i» elementi
        "errore: la lista è troppo corta"
        ; altrimenti procede
        (if (= i 1)
            ; se si tratta del primo elemento, basta la funzione
            ; car per prelevarlo
            (car x)
            ; altrimenti, si utilizza una chiamata ricorsiva
            (i-esimo-elemento (- i 1) (cdr x))
        )
    )
)
```

- Estrae l'ultimo elemento:

```
(define (ultimo x)
    (if (null? x)
        ; la lista è vuota e questo è un errore
        "errore: la lista è vuota"
        ; altrimenti si occupa di estrarre l'ultimo elemento
        (if (null? (cdr x))
            ; se si tratta di una lista contenente un solo elemento,
            ; restituisce il primo e unico di questa
            (car x)
            ; altrimenti utilizza una chiamata ricorsiva
            (ultimo (cdr x))
        )
    )
)
```

- Elimina l'ultimo elemento:

```
(define (elimina-ultimo x)
    (if (null? x)
        ; la lista è vuota e questo è un errore
        "errore: la lista è vuota"
        ; altrimenti si occupa di eliminare l'ultimo elemento
        (if (null? (cdr x))
```

```
            ; se si tratta di una lista contenente un solo elemento,
            ; restituisce la lista vuota
            '()
            ; altrimenti utilizza una chiamata ricorsiva per comporre
            ; una lista senza l'ultimo elemento
            (cons (car x) (elimina-ultimo (cdr x)))
        )
    )
)
```

- Restituisce la parte finale della lista, escludendo alcuni elementi iniziali. Si tratta precisamente di una funzione di libreria di Scheme, denominata '**`list-tail`**':

```
(define (list-tail x k)
    (if (zero? k)
        ; se «k» è pari a zero, viene restituita tutta la lista
        x
        ; altrimenti occorre eliminare k-1 elementi iniziali
        ; da (cdr x)
        (list-tail (cdr x) (- k 1))
    )
)
```

- Ricerca del (k+1)-esimo elemento di una lista. Si tratta di una funzione di libreria di Scheme, denominata '**`list-ref`**' (in pratica, l'indice k viene usato in modo da indicare il primo elemento con il numero zero):

```
(define (list-ref x k)
    ; si limita a restituire il primo elemento ottenuto
    ; dalla funzione list-tail
    (car (list-tail x k))
)
```

- Scansione di una lista in modo da restituire un'altra lista, contenente i valori restituiti dalla chiamata di una funzione data per ogni elemento della lista. Si tratta di una semplificazione della funzione di libreria '**`map`**', in questo caso con la possibilità di indicare una sola lista di valori di partenza:

```
(define (map1 f x)
    ; «f» è la funzione da applicare agli elementi della lista «x»
    (if (null? x)
        ; la lista è vuota e restituisce un'altra lista vuota
        '()
        ; altrimenti compone la lista da restituire
        (cons (f (car x)) (map1 f (cdr x)))
    )
)
```

- Descrizione della funzione di libreria '**`append`**':

```
(define (append x y)
    (if (null? x)
        ; se la lista «x» è vuota, restituisce la lista «y»
        y
        ; altrimenti costruisce la lista in modo ricorsivo
        (cons
            (car x)
            (append (cdr x) y)
        )
    )
)
```

- Descrizione della funzione di libreria '**`reverse`**':

```
(define (reverse x)
    (if (null? x)
        ; se la lista «x» è vuota, non c'è nulla da invertire
        '()
        ; altrimenti compone l'inversione con una chiamata ricorsiva
        (append (reverse (cdr x)) (list (car x)))
    )
)
```

308.2 Vettori

Scheme gestisce anche i vettori, che sono in pratica gli array dei linguaggi di programmazione normali. Un vettore viene rappresentato in forma costante come una lista preceduta dal simbolo '#':

```
#( elemento_1 ... elemento_n )
```

L'indice dei vettori di Scheme parte da zero. Il funzionamento dei vettori di Scheme non richiede spiegazioni particolari. La tabella 308.2 riassume le funzioni utili con questo tipo di dati.

Tabella 308.2. Elenco di alcune funzioni specifiche per la gestione dei vettori.

Funzione	Descrizione
(vector? *oggetto*)	*Vero* se l'oggetto è un vettore.
(make-vector *k*)	Restituisce un vettore di *k* elementi indefiniti.
(make-vector *k valore*)	Restituisce un vettore di *k* elementi inizializzati al valore specificato.
(vector *elemento* ...)	Restituisce un vettore degli elementi indicati.
(vector-length *vettore*)	Restituisce il numero di elementi del vettore.
(vector-ref *vettore k*)	Restituisce l'elemento nella posizione *k*, partendo da zero.
(vector-set! *vettore k oggetto*)	Assegna all'elemento *k*-esimo l'oggetto indicato.
(vector->list *vettore*)	Converte il vettore in lista.
(list->vector *lista*)	Converte la lista in vettore.

308.3 Strutture di controllo applicate alle liste

Alcune funzioni tipiche di Scheme servono ad applicare una funzione a un gruppo di valori contenuto in una lista.

Tabella 308.3. Elenco di alcune funzioni specifiche per la scansione degli elementi di una lista, allo scopo di applicarvi una funzione.

Funzione	Descrizione
(apply *funzione lista*)	Esegue la funzione utilizzando gli elementi della lista come argomenti.
(map *funzione lista* ...)	Esegue la funzione iterativamente per gli elementi delle liste.
(for-each *funzione lista* ...)	Esegue la funzione iterativamente per gli elementi delle liste.

308.3.1 apply

```
(apply funzione lista )
```

La funzione '**apply**' esegue una funzione a cui affida gli elementi di una lista come altrettanti argomenti. In pratica,

```
(apply funzione '(elem_1 elem_2 ... elem_n ))
```

equivale a:

```
(funzione elem_1 elem_2 ... elem_n )
```

Per esempio:

```
(apply + '(1 2))                ===> 3
```

308.3.2 map

`(map` *funzione lista …*`)`

La funzione '**`map`**' scandisce una o più liste, tutte con la stessa quantità di elementi, in modo tale che, a ogni ciclo, viene passato alla funzione l'insieme ordinato dell'*i*-esimo elemento di ognuna di queste liste. La funzione restituisce una lista contenente i valori restituiti dalla funzione a ogni ciclo.

Anche se viene rispettato l'ordine delle varie liste, '**`dat`**' non garantisce che la scansione avvenga dal primo elemento all'ultimo.

L'esempio seguente esegue la somma di una serie di coppie di valori, restituendo la lista dei risultati:

```
(map + '(1 2 3) '(4 5 6))               ===> (5 7 9)
```

308.3.3 for-each

`(for-each` *funzione lista …*`)`

La funzione '**`for-each`**' è molto simile a '**`map`**', nel senso che avvia una funzione ripetutamente, quanti sono gli elementi delle liste successive, garantendo di eseguire l'operazione in ordine, secondo la sequenza degli elementi nelle liste. Tuttavia, non restituisce nulla.

308.4 Riferimenti

- A. Aaby, *Scheme Tutorial*, 1996

  <http://cs-sa1.wwc.edu/~cs_dept/KU/PR/Scheme.html>

- *R^{5}RS -- Revised-5 Report on the Algorithmic Language Scheme*, 1998

  <http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html>

  <http://www.swiss.ai.mit.edu/ftpdir/scheme-reports/r5rs.ps.gz>

Scheme: I/O

Scheme ha una gestione particolare dei file. Per prima cosa, i flussi di file, che negli altri linguaggi sono dei *file handle*, in Scheme prendono il nome di *port*, *porte*. Scheme distingue quindi tra porte in ingresso, in grado di «consegnare» dei caratteri, e porte in uscita, in grado di «accettare» caratteri.

309.1 Apertura e chiusura

Scheme distingue tra flussi di file in ingresso e in uscita, per cui le funzioni per aprire i file e trasformarli in porte, sono due, uno per l'apertura in lettura (ingresso) e l'altra per l'apertura in scrittura (uscita). La tabella 309.1 riassume le funzioni utili per aprire, controllare e chiudere i file. Gli esempi successivi, dovrebbero aiutare a comprenderne l'utilizzo.

Tabella 309.1. Elenco di alcune funzioni per l'apertura e la chiusura dei file, oltre che per il controllo dei flussi di file predefiniti.

Funzione	Descrizione
(open-input-file *str_nome_file*)	Apre il file nominato e restituisce la porta in ingresso.
(open-output-file *str_nome_file*)	Apre il file nominato e restituisce la porta in uscita.
(port? *oggetto*)	*Vero* se si tratta di una porta.
(input-port? *oggetto*)	*Vero* se si tratta di una porta in ingresso.
(output-port? *oggetto*)	*Vero* se si tratta di una porta in uscita.
(close-input-port *porta*)	Chiude la porta in ingresso.
(close-output-port *porta*)	Chiude la porta in uscita.

```
(define porta-i (open-input-file "mio_file"))

(port? porta-i)                                 ===> #t
(output-port? porta-i)                          ===> #f
(input-port? porta-i)                           ===> #t

(close-input-port porta-i)
```

In condizioni normali, sono sempre disponibili una porta in ingresso e una in uscita, in modo predefinito. Si tratta generalmente di standard input e standard output. Questi flussi di file predefiniti potrebbero essere diretti verso altri file. Tuttavia questo non viene mostrato; eventualmente si può approfondire il problema leggendo *R*^{5}*RS*.

309.2 Ingresso dei dati

L'ingresso dei dati, ovvero la lettura, avviene attraverso due funzioni fondamentali: '**`read-char`**' e '**`read`**'. La prima legge un carattere alla volta, la seconda interpreta ciò che legge in forma di dati Scheme. In pratica, '**`read`**' legge ogni volta ciò che riesce a interpretare come un oggetto per Scheme.

Tabella 309.2. Elenco di alcune funzioni per la gestione dei dati in ingresso.

Funzione	Descrizione
(read-char)	Legge e restituisce il carattere successivo dalla porta predefinita.
(read-char *porta*)	Legge e restituisce il carattere successivo dalla porta indicata.
(peek-char)	Restituisce una copia del carattere successivo dalla porta predefinita.
(peek-char *porta*)	Restituisce una copia del carattere successivo dalla porta indicata.
(read)	Legge un oggetto dalla porta predefinita.
(read *porta*)	Legge un oggetto dalla porta indicata.
(eof-object *porta*)	*Vero* la lettura dalla porta ha raggiunto la fine.

L'esempio seguente mostra in che modo potrebbe essere utilizzata la funzione '**`read-char`**'. Si inizia aprendo il file '`/etc/passwd`', dal quale vengono letti i primi caratteri. Si suppone che il primo record a essere letto sia quello di definizione dell'utente '**`root`**'.

```
; apre il file e gli associa la porta «utenti»
(define utenti (open-input-file "/etc/passwd"))

; legge un carattere alla volta
(read-char utenti)                              ===> #\r
(read-char utenti)                              ===> #\o
(read-char utenti)                              ===> #\o
(read-char utenti)                              ===> #\t
(read-char utenti)                              ===> #\:
;...

; chiude il file
(close-input-file utenti)
```

Nell'esempio seguente si vuole mostrare l'uso della funzione '**`read`**'. Prima si suppone di avere preparato il file seguente:

```
; prova_lettura.scm

; somma
(+ 1 2)

; moltiplicazione
(* 2 5)

; stringa
"ciao"

; valore numerico
123

; fine
```

Supponendo che il file si chiami '`prova_lettura.scm`', si osservi la sequenza di istruzioni Scheme seguente, assieme a ciò che si ottiene dalla lettura del file:

```
; apre il file e gli associa la porta «prova»
(define prova (open-input-file "prova_lettura.scm"))

; legge il primo oggetto
(read utenti)                                   ===> (+ 1 2)

; legge il secondo oggetto
(read utenti)                                   ===> (* 2 5)

; legge il terzo oggetto
(read utenti)                                   ===> "ciao"

; legge il quarto oggetto
(read utenti)                                   ===> 123
```

```
; chiude il file
(close-input-file prova)
```

Si intende l'importanza della funzione '**read**' per facilitare l'inserimento di dati nei programmi in modo interattivo.

309.3 Uscita dei dati

L'emissione dei dati, ovvero la scrittura, avviene in maniera simile alla lettura, con la stessa distinzione tra le funzioni '**write-char**' e '**write**'. Anche in questo caso, la prima scrive un carattere alla volta, mentre la seconda emette la rappresentazione di un oggetto alla volta. Tuttavia, si aggiunte un'altra funzione fondamentale: '**output**'. Questa funzione viene usata preferibilmente per mostrare dei messaggi senza codici di escape, soprattutto per non lasciare le virgolette di delimitazione delle stringhe.

Tabella 309.3. Elenco di alcune funzioni per la gestione dei dati in ingresso.

Funzione	Descrizione
(write-char *carattere*)	Scrive il carattere indicato attraverso la porta predefinita.
(write-char *carattere porta*)	Scrive il carattere indicato attraverso la porta indicata.
(write *oggetto*)	Scrive la rappresentazione dell'oggetto attraverso la porta predefinita.
(write *oggetto porta*)	Scrive la rappresentazione dell'oggetto attraverso la porta indicata.
(display *oggetto*)	Mostra l'oggetto attraverso la porta predefinita.
(display *oggetto porta*)	Mostra l'oggetto attraverso la porta indicata.
(newline)	Emette un codice di interruzione di riga attraverso la porta predefinita.
(newline *porta*)	Emette un codice di interruzione di riga attraverso la porta indicata.

L'esempio seguente dovrebbe chiarire la differenza tra la funzione '**write**' e '**display**'. Gli oggetti vengono emessi attraverso lo standard output, ovvero la porta predefinita.

```
(write (+ 1 2))                     ; visualizza «3»
(write "ciao")                      ; visualizza «"ciao"»
(write "ciao, come \"stai\"?")      ; visualizza «"ciao, come \"stai\""»
(write #\A)                         ; visualizza «#\A»
(display (+ 1 2))                   ; visualizza «3»
(display "ciao")                    ; visualizza «ciao»
(display "ciao, come \"stai\"?")    ; visualizza «ciao, come "stai"»
(display #\A)                       ; visualizza «A»
```

È già stato descritto l'uso di '**newline**', che è indispensabile per ottenere l'avanzamento alla riga successiva. In linea di principio, non è possibile inserire un carattere di controllo nella stringa emessa da '**write**' o da '**display**'.

Scheme: esempi di programmazione

Questo capitolo raccoglie solo alcuni esempi di programmazione, in parte già descritti in altri capitoli. Lo scopo di questi esempi è solo didattico, utilizzando forme non ottimizzate per la velocità di esecuzione.

310.1 Problemi elementari di programmazione 3502

310.1.1 Somma tra due numeri positivi 3502
310.1.2 Moltiplicazione di due numeri positivi attraverso la somma 3504
310.1.3 Divisione intera tra due numeri positivi 3505
310.1.4 Elevamento a potenza 3506
310.1.5 Radice quadrata 3507
310.1.6 Fattoriale 3507
310.1.7 Massimo comune divisore 3508
310.1.8 Numero primo 3509

310.2 Scansione di array 3510

310.2.1 Ricerca sequenziale 3510
310.2.2 Ricerca binaria 3511

310.3 Algoritmi tradizionali 3512

310.3.1 Bubblesort 3512
310.3.2 Torre di Hanoi 3514
310.3.3 Quicksort 3515
310.3.4 Permutazioni 3517

310.1 Problemi elementari di programmazione

In questa sezione vengono mostrati alcuni algoritmi elementari portati in Scheme. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

310.1.1 Somma tra due numeri positivi

Il problema della somma tra due numeri positivi, attraverso l'incremento unitario, è stato descritto nella sezione 282.2.1.

```
; ======================================================================
; somma1.scm
; Somma esclusivamente valori positivi.
; ======================================================================

; ======================================================================
; (somma <x> <y>)
; ----------------------------------------------------------------------
(define (somma x y)
    (define z x)
```

```
    (define i 1)

    (do ()
        ((> i y))

        (set! z (+ z 1))
        (set! i (+ i 1))
    )

    z
)

; =====================================================================
; Inizio del programma.
; ---------------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (somma x y))
(display x) (display " + ") (display y) (display " = ") (display z)
(newline)

; =====================================================================
```

In alternativa, si può modificare la funzione '**somma**', in modo che il ciclo '**do**' gestisca la dichiarazione e l'incremento delle variabili che usa. Tuttavia, in questo caso, la variabile '**z**' deve essere «copiata» in modo da poter trasmettere il risultato all'esterno del ciclo '**do**'.

```
(define (somma x y)
    (define risultato 0)

    (do ((z x (+ z 1)) (i 1 (+ i 1)))
        ((> i y))
        (set! risultato z)
    )

    risultato
)
```

Volendo gestire la cosa in modo un po' più elegante, occorre togliere la variabile '**z**' dalla gestione del ciclo '**do**':

```
(define (somma x y)
    (define z x)

    (do ((i 1 (+ i 1)))
        ((> i y))
        (set! z (+ z 1))
    )

    z
)
```

310.1.2 Moltiplicazione di due numeri positivi attraverso la somma

Il problema della moltiplicazione tra due numeri positivi, attraverso la somma, è stato descritto nella sezione 282.2.2.

```
; =====================================================================
; moltiplica1.scm
; Moltiplica esclusivamente valori positivi.
; =====================================================================

; =====================================================================
; (moltiplica <x> <y>)
; ---------------------------------------------------------------------
(define (moltiplica x y)
    (define z 0)
    (define i 1)

    (do ()
        ((> i y))

        (set! z (+ z x))
        (set! i (+ i 1))
    )

    z
)

; =====================================================================
; Inizio del programma.
; ---------------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (moltiplica x y))
(display x) (display " * ") (display y) (display " = ") (display z)
(newline)

; =====================================================================
```

In alternativa, si può modificare la funzione '**`moltiplica`**', in modo che il ciclo '**`do`**' gestisca la dichiarazione e l'incremento dell'indice '**`i`**'.

```
(define (moltiplica x y)
    (define z 0)

    (do ((i 1 (+ i 1)))
        ((> i y))

        (set! z (+ z x))
    )

    z
)
```

310.1.3 Divisione intera tra due numeri positivi

Il problema della divisione tra due numeri positivi, attraverso la sottrazione, è stato descritto nella sezione 282.2.3.

```
; ======================================================================
; dividi1.scm
; Divide esclusivamente valori positivi.
; ======================================================================

; ======================================================================
; (dividi <x> <y>)
; ----------------------------------------------------------------------
(define (dividi x y)
    (define z 0)
    (define i x)

    (do ()
        ((< i y))

        (set! i (- i y))
        (set! z (+ z 1))
    )

    z
)

; ======================================================================
; Inizio del programma.
; ----------------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (dividi x y))
(display x) (display " / ") (display y) (display " = ") (display z)
(newline)

; ======================================================================
```

In alternativa, si può modificare la funzione '**dividi**', in modo che il ciclo '**do**' gestisca la dichiarazione e il decremento della variabile '**i**'. Per la precisione, la variabile '**z**' non può essere dichiarata nello stesso modo, perché serve anche al di fuori del ciclo.

```
(define (dividi x y)
    (define z 0)

    (do ((i x (- i y)))
        ((< i y))

        (set! z (+ z 1))
    )

    z
)
```

310.1.4 Elevamento a potenza

Il problema dell'elevamento a potenza tra due numeri positivi, attraverso la moltiplicazione, è stato descritto nella sezione 282.2.4.

```
; ======================================================================
; potenza1.scm
; Eleva a potenza.
; ======================================================================

; ======================================================================
; (potenza <x> <y>)
; ----------------------------------------------------------------------
(define (potenza x y)
    (define z 1)
    (define i 1)

    (do ()
        ((> i y))

        (set! z (* z x))
        (set! i (+ i 1))
    )

    z
)

; ======================================================================
; Inizio del programma.
; ----------------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (potenza x y))
(display x) (display " ** ") (display y) (display " = ") (display z)
(newline)

; ======================================================================
```

In alternativa, si può modificare la funzione '**potenza**', in modo che il ciclo '**do**' gestisca la dichiarazione e l'incremento della variabile '**i**'.

```
(define (potenza x y)
    (define z 1)

    (do ((i 1 (+ i 1)))
        ((> i y))

        (set! z (* z x))
    )

    z
)
```

È possibile usare anche un algoritmo ricorsivo.

```
(define (potenza x y)
    (if (= x 0)
        0
```

```
        (if (= y 0)
            1
            (* x (potenza x (- y 1)))
        )
    )
)
```

310.1.5 Radice quadrata

Il problema della radice quadrata è stato descritto nella sezione 282.2.5.

```
; ======================================================================
; radice1.scm
; Radice quadrata.
; ======================================================================

; ======================================================================
; (radice <x>)
; ----------------------------------------------------------------------
(define (radice x)
    (define z -1)
    (define t 0)
    (define uscita #f)

    (do ()
        (uscita)

        (set! z (+ z 1))
        (set! t (* z z))
        (if (> t x)
            ; È stato superato il valore massimo
            (begin
                (set! z (- z 1))
                (set! uscita #t)
            )
        )
    )

    z
)

; ======================================================================
; Inizio del programma.
; ----------------------------------------------------------------------
(define x 0)
(define z 0)

(display "Inserisci il numero intero positivo: ")
(set! x (read))
(newline)
(set! z (radice x))
(display "La radice quadrata di ") (display x) (display " è ") (display z)
(newline)

; ======================================================================
```

310.1.6 Fattoriale

Il problema del fattoriale è stato descritto nella sezione 282.2.6.

```
; ======================================================================
; fattoriale1.scm
; Fattoriale.
; ======================================================================
```

```
; ======================================================================
; (fattoriale <x>)
; ----------------------------------------------------------------------
(define (fattoriale x)
    (define i (- x 1))

    (do ()
        ((<= i 0))

        (set! x (* x i))
        (set! i (- i 1))
    )

    x
)

; ======================================================================
; Inizio del programma.
; ----------------------------------------------------------------------
(define x 0)
(define z 0)

(display "Inserisci il numero intero positivo: ")
(set! x (read))
(newline)
(set! z (fattoriale x))
(display x) (display "! = ") (display z)
(newline)

; ======================================================================
```

In alternativa, l'algoritmo si può tradurre in modo ricorsivo.

```
(define (fattoriale x)
    (if (> x 1)
        (* x (fattoriale (- x 1)))
        1
    )
)
```

310.1.7 Massimo comune divisore

Il problema del massimo comune divisore, tra due numeri positivi, è stato descritto nella sezione 282.2.7.

```
; ======================================================================
; mcd1.scm
; Massimo Comune Divisore.
; ======================================================================

; ======================================================================
; (moltiplica <x> <y>)
; ----------------------------------------------------------------------
(define (mcd x y)
    (do ()
        ((= x y))

        (if (> x y)
            (set! x (- x y))
            (set! y (- y x))
        )
    )

    x
)
```

```
; ======================================================================
; Inizio del programma.
; ----------------------------------------------------------------------
(define x 0)
(define y 0)
(define z 0)

(display "Inserisci il primo numero intero positivo: ")
(set! x (read))
(newline)
(display "Inserisci il secondo numero intero positivo: ")
(set! y (read))
(newline)
(set! z (mcd x y))
(display "MCD di ") (display x) (display " e ") (display y)
(display " è ") (display z)
(newline)

; ======================================================================
```

310.1.8 Numero primo

Il problema della determinazione se un numero sia primo o meno, è stato descritto nella sezione 282.2.8.

```
; ======================================================================
; primo1.scm
; Numero primo.
; ======================================================================

; ======================================================================
; (primo <x>)
; ----------------------------------------------------------------------
(define (primo x)
    (define np #t)
    (define i 2)
    (define j 0)

    (do ()
        ((or (>= i x) (not np)))

        (set! j (truncate (/ x i)))
        (set! j (- x (* j i)))
        (if (= j 0)
            (set! np #f)
            (set! i (+ i 1))
        )
    )

    np
)

; ======================================================================
; Inizio del programma.
; ----------------------------------------------------------------------
(define x 0)

(display "Inserisci un numero intero positivo: ")
(set! x (read))
(newline)
(if (primo x)
    (display "È un numero primo")
    (display "Non è un numero primo")
)
```

```
(newline)

; ======================================================================
```

310.2 Scansione di array

In questa sezione vengono mostrati alcuni algoritmi, legati alla scansione degli array, portati in Scheme, dove vengono usati i vettori di questo linguaggio. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

310.2.1 Ricerca sequenziale

Il problema della ricerca sequenziale all'interno di un array, è stato descritto nella sezione 282.3.1.

```
; ======================================================================
; ricerca_sequenziale1.scm
; Ricerca Sequenziale.
; ======================================================================

; ======================================================================
; (ricerca <vettore> <x> <ele-inf> <ele-sup>)
; ----------------------------------------------------------------------
(define (ricerca vettore x a z)
    (define risultato -1)

    (do ((i a (+ i 1)))
        ((> i z))

        (if (= x (vector-ref vettore i))
            (set! risultato i)
        )
    )
    risultato
)

; ======================================================================
; Inizio del programma.
; ----------------------------------------------------------------------

(define DIM 100)
(define vettore (make-vector DIM))
(define x 0)
(define i 0)
(define z 0)

(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore.")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
```

```
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

(display "Inserire il valore da cercare: ")
(set! x (read))
(newline)

(set! i (ricerca vettore x 0 (- z 1)))

(display "Il valore cercato si trova nell'elemento ")
(display i)
(newline)

; ================================================================
```

Esiste anche una soluzione ricorsiva che viene mostrata di seguito:

```
(define (ricerca vettore x a z)
    (if (> a z)
        ; La corrispondenza non è stata trovata.
        1
        (if (= x (vector-ref vettore a))
            a
            (ricerca vettore x (+ a 1) z)
        )
    )
)
```

310.2.2 Ricerca binaria

Il problema della ricerca binaria all'interno di un array, è stato descritto nella sezione 282.3.2.

```
; ================================================================
; ricerca_binaria1.scm
; Ricerca Binaria.
; ================================================================

; ================================================================
; (ricerca <vettore> <x> <ele-inf> <ele-sup>)
; ----------------------------------------------------------------
(define (ricerca vettore x a z)
    (define m (truncate (/ (+ a z) 2)))

    (if (or (< m a) (> m z))
        ; Non restano elementi da controllare: l'elemento cercato
        ; non c'è.
        -1

        (if (< x (vector-ref vettore m))
            ; Si ripete la ricerca nella parte inferiore.
            (ricerca vettore x a (- m 1))

            (if (> x (vector-ref vettore m))
                ; Si ripete la ricerca nella parte superiore.
                (ricerca vettore x (+ m 1) z)

                ; Se x è uguale a vettore[m], l'obiettivo è
                ; stato trovato.
                m
            )
        )
    )
)
```

```
; ======================================================================
; Inizio del programma.
; ----------------------------------------------------------------------

(define DIM 100)
(define vettore (make-vector DIM))
(define x 0)
(define i 0)
(define z 0)

(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore (in modo ordinato).")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

(display "Inserire il valore da cercare: ")
(set! x (read))
(newline)

(set! i (ricerca vettore x 0 (- z 1)))

(display "Il valore cercato si trova nell'elemento ")
(display i)
(newline)

; ======================================================================
```

310.3 Algoritmi tradizionali

In questa sezione vengono mostrati alcuni algoritmi tradizionali portati in Scheme. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

310.3.1 Bubblesort

Il problema del Bubblesort è stato descritto nella sezione 282.4.1. Viene mostrato prima una soluzione iterativa e successivamente la funzione '**`bsort`**' in versione ricorsiva.

```
; ======================================================================
; bsort1.scm
; Bubblesort.
; ======================================================================

; ======================================================================
; (ordina <vettore> <ele-inf> <ele-sup>)
; ----------------------------------------------------------------------
```

```
(define (ordina vettore a z)
    (define scambio 0)

    (do ((j a (+ j 1)))
        ((>= j z))

        (do ((k (+ j 1) (+ k 1)))
            ((> k z))

            (if (< (vector-ref vettore k) (vector-ref vettore j))
                ; Scambia i valori.
                (begin
                    (set! scambio (vector-ref vettore k))
                    (vector-set! vettore k (vector-ref vettore j))
                    (vector-set! vettore j scambio)
                )
            )
        )
    )

    vettore
)

; ============================================================================
; Inizio del programma.
; ----------------------------------------------------------------------------

(define DIM 100)
(define vettore (make-vector DIM))
(define x 0)
(define i 0)
(define z 0)

(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore.")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

(set! vettore (ordina vettore 0 (- z 1)))

(display "Il vettore ordinato è il seguente: ")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display (vector-ref vettore i))
    (display " ")
)
(newline)
```

```
; ======================================================================
```

Segue la funzione '**ordina**' in versione ricorsiva.

```
(define (ordina vettore a z)
    (define scambio 0)

    (if (< a z)
        (begin
            ; Scansione interna dell'array per collocare nella
            ; posizione a l'elemento giusto.
            (do ((k (+ a 1) (+ k 1)))
                ((> k z))

                (if (< (vector-ref vettore k) (vector-ref vettore a))
                    ; Scambia i valori.
                    (begin
                        (set! scambio (vector-ref vettore k))
                        (vector-set! vettore k (vector-ref vettore a))
                        (vector-set! vettore a scambio)
                    )
                )
            )

            (set! vettore (ordina vettore (+ a 1) z))
        )
    )
    vettore
)
```

310.3.2 Torre di Hanoi

Il problema della torre di Hanoi è stato descritto nella sezione 282.4.2.

```
; ======================================================================
; hanoi1.scm
; Torre di Hanoi.
; ======================================================================

; ======================================================================
; (hanoi <n-anelli> <piolo-iniziale> <piolo-finale>)
; ----------------------------------------------------------------------
(define (hanoi n p1 p2)
    (if (> n 0)
        (begin
            (hanoi (- n 1) p1 (- 6 (+ p1 p2)))
            (begin
                (display "Muovi l'anello ")
                (display n)
                (display " dal piolo ")
                (display p1)
                (display " ")
                (display p2)
                (newline)
            )
            (hanoi (- n 1) (- 6 (+ p1 p2)) p2)
        )
    )
)


; ======================================================================
; Inizio del programma.
; ----------------------------------------------------------------------
(define n 0)
```

```
(define p1 0)
(define p2 0)

(display "Inserisci il numero di pioli: ")
(set! n (read))
(newline)
(display "Inserisci il numero del piolo iniziale (da 1 a 3): ")
(set! p1 (read))
(newline)
(display "Inserisci il numero del piolo finale (da 1 a 3): ")
(set! p2 (read))
(newline)
(hanoi n p1 p2)

; ============================================================================
```

310.3.3 Quicksort

L'algoritmo del Quicksort è stato descritto nella sezione 282.4.3.

```
; ============================================================================
; qsort1.scm
; Quicksort.
; ============================================================================

; ----------------------------------------------------------------------------
; Dichiara il vettore a cui faranno riferimento tutte le funzioni.
; Il vettore non viene passato alle funzioni tra gli argomenti, per
; semplificare le funzioni, soprattutto nel caso di «part», che
; deve restituire anche un altro valore.
; ----------------------------------------------------------------------------
(define DIM 100)
(define vettore (make-vector DIM))

; ============================================================================
; (inverti-elementi <indice-1> <indice-2>)
; ----------------------------------------------------------------------------
(define (inverti-elementi a z)
    (define scambio 0)
    (set! scambio (vector-ref vettore a))
    (vector-set! vettore a (vector-ref vettore z))
    (vector-set! vettore z scambio)
)

; ============================================================================
; (part <ele-inf> <ele-sup>)
; ----------------------------------------------------------------------------
(define (part a z)
    ; Si assume che «a» sia inferiore a «z».
    (define i (+ a 1))
    (define cf z)
    ; Vengono preparate delle variabili per controllare l'uscita dai cicli.
    (define uscita1 #f)
    (define uscita2 #f)
    (define uscita3 #f)

    ; Inizia il ciclo di scansione dell'array.
    (set! uscita1 #f)
    (do ()
        (uscita1)
        (set! uscita2 #f)
        (do ()
            (uscita2)

            ; Sposta «i» a destra.
```

```
            (if (or
                    (> (vector-ref vettore i) (vector-ref vettore a))
                    (>= i cf)
                )
                ; Interrompe il ciclo interno.
                (set! uscita2 #t)
                ; Altrimenti incrementa l'indice
                (set! i (+ i 1))
            )
        )
        (set! uscita3 #f)
        (do ()
            (uscita3)

            ; Sposta «cf» a sinistra.
            (if (<= (vector-ref vettore cf) (vector-ref vettore a))
                ; Interrompe il ciclo interno.
                (set! uscita3 #t)
                ; Altrimenti decrementa l'indice
                (set! cf (- cf 1))
            )
        )

        (if (<= cf i)
            ; È avvenuto l'incontro tra «i» e «cf».
            (set! uscita1 #t)
            ; Altrimenti vengono scambiati i valori.
            (begin
                (inverti-elementi i cf)
                (set! i (+ i 1))
                (set! cf (- cf 1))
            )
        )
    )

    ; A questo punto vettore[a..z] è stato ripartito e «cf» è la
    ; collocazione di vettore[a].
    (inverti-elementi a cf)

    ; A questo punto, vettore[cf] è un elemento (un valore) nella
    ; posizione giusta, e «cf» è ciò che viene restituito.
    cf
)

; ============================================================================
; (ordina <ele-inf> <ele-sup>)
; ----------------------------------------------------------------------------
(define (ordina a z)
    ; Viene preparata la variabile «cf».
    (define cf 0)

    (if (> z a)
        (begin
            (set! cf (part a z))
            (ordina a (- cf 1))
            (ordina (+ cf 1) z)
        )
    )
)

; ============================================================================
; Inizio del programma.
; ----------------------------------------------------------------------------

(define x 0)
(define i 0)
```

```
(define z 0)

(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! z (read))
(newline)

(if (> z DIM)
    (set! z DIM)
)

(display "Inserire i valori del vettore.")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

; Il vettore non viene trasferito come argomento della funzione,
; ma risulta accessibile esternamente.
(ordina 0 (- z 1))

(display "Il vettore ordinato è il seguente: ")
(newline)
(do ((i 0 (+ i 1)))
    ((>= i z))

    (display (vector-ref vettore i))
    (display " ")
)
(newline)

; ========================================================================
```

310.3.4 Permutazioni

L’algoritmo ricorsivo delle permutazioni è stato descritto nella sezione 282.4.4.

```
; ========================================================================
; permuta1.scm
; Permutazioni.
; ========================================================================

; ------------------------------------------------------------------------
; Dichiara il vettore a cui faranno riferimento tutte le funzioni.
; ------------------------------------------------------------------------
(define DIM 100)
(define vettore (make-vector DIM))

; ------------------------------------------------------------------------
; Sempre per motivi pratici, rende disponibile la dimensione utilizzata
; effettivamente.
; ------------------------------------------------------------------------
(define n-elementi 0)

; ========================================================================
; (inverti-elementi <indice-1> <indice-2>)
; ------------------------------------------------------------------------
(define (inverti-elementi a z)
```

```
    (define scambio 0)
    (set! scambio (vector-ref vettore a))
    (vector-set! vettore a (vector-ref vettore z))
    (vector-set! vettore z scambio)
)

; ========================================================================
; (visualizza)
; ------------------------------------------------------------------------
(define (visualizza)
    (do ((i 0 (+ i 1)))
        ((>= i n-elementi))

        (display (vector-ref vettore i))
        (display " ")
    )
    (newline)
)

; ========================================================================
; (permuta <inizio> <fine>)
; ------------------------------------------------------------------------
(define (permuta a z)
    (define k 0)

    ; Se il segmento di array contiene almeno due elementi, si
    ; procede.
    (if (>= (- z a) 1)
        ; Inizia un ciclo di scambi tra l'ultimo elemento e uno
        ; degli altri contenuti nel segmento di array.
        (do ((k z (- k 1)))
            ((< k a))

            ; Scambia i valori.
            (inverti-elementi k z)

            ; Esegue una chiamata ricorsiva per permutare un
            ; segmento più piccolo dell'array.
            (permuta a (- z 1))

            ; Scambia i valori.
            (inverti-elementi k z)
        )

        ; Altrimenti, visualizza l'array e utilizza una variabile
        ; dichiarata globalmente.
        (visualizza)
    )
)

; ========================================================================
; Inizio del programma.
; ------------------------------------------------------------------------
(display "Inserire la quantità di elementi; ")
(display DIM)
(display " al massimo: ")
(set! n-elementi (read))
(newline)

(if (> n-elementi DIM)
    (set! n-elementi DIM)
)

(display "Inserire i valori del vettore.")
(newline)
(do ((i 0 (+ i 1)))
```

```
    ((>= i n-elementi))

    (display "elemento ")
    (display i)
    (display " ")
    (vector-set! vettore i (read))
    (newline)
)

; Il vettore non viene trasferito come argomento della funzione,
; ma risulta accessibile esternamente.
(permuta 0 (- n-elementi 1))

; ======================================================================
```

Parte lx

BC: linguaggio aritmetico a precisione arbitraria

311 BC 3522

311.1 Base di numerazione 3522
311.2 Approssimazione 3523
311.3 Linguaggio di programmazione 3524
311.4 Utilizzo di BC 3529
311.5 BC nella realizzazione GNU 3530
311.6 Riferimenti 3530

312 BC: esempi di programmazione 3531

312.1 Problemi elementari di programmazione 3532
312.2 Scansione di array 3536
312.3 Algoritmi tradizionali 3537

BC

BC è un interprete di un linguaggio aritmetico, che fa parte della tradizione dei sistemi Unix, tanto da essere codificato anche nello standard POSIX. Come linguaggio non ha nulla di speciale, ma la sua facilità di utilizzo in modo interattivo e la sua diffusione, lo rendono molto comodo e utile.

L'utilizzo più conveniente di BC è probabilmente quello a riga di comando, come calcolatrice, tenendo conto che questa sua caratteristica può anche essere sfruttata utilmente all'interno di script di shell. L'esempio seguente mostra un utilizzo interattivo, per comprendere di cosa si tratta, almeno a prima vista:

```
$ bc[ Invio ]

255*63*3737*512[ Invio ]

30737871360

[ Ctrl+d ]
```

Quello che si vede nell'esempio è la moltiplicazione di tre numeri: 255, 63, 3737 e 512. Il risultato è ciò che si vede alla fine: 30737871360. La stessa cosa si poteva inserire in uno script di shell nel modo seguente, in cui il risultato della moltiplicazione viene assegnato alla variabile '**RISULTATO**':

```
RISULTATO=‘echo "255*63*3737*512" | bc‘
```

Tuttavia, BC è in realtà un linguaggio di programmazione, benché semplice, la cui caratteristica fondamentale è quella di poter definire l'approssimazione del risultato, indipendentemente dall'architettura dell'elaboratore per il quale è stato compilato.

311.1 Base di numerazione

Una caratteristica importante di BC è la possibilità di gestire basi di numerazione diverse da 10. Tuttavia, ciò può creare degli imprevisti inattesi, per cui occorre fare attenzione quando si tenta di modificare la convenzione normale.

La base di numerazione viene modificata intervenendo attraverso due variabili predefinite che fanno parte del linguaggio, denominate '**ibase**' e '**obase**'. La prima contiene la base di numerazione per i numeri che vengono inseriti, mentre la seconda contiene la base usata per la rappresentazione dei risultati. In condizioni normali, sia '**ibase**', sia '**obase**', contengono il valore 10. Tuttavia, quando si cambia il valore di '**ibase**' si possono creare delle complicazioni; supponendo di voler inserire valori in base otto, basta agire come segue:

```
ibase=8[ Invio ]
```

Nel momento in cui si scrive un valore, questo viene interpretato in base otto:

```
777[ Invio ]

511
```

Infatti, 777_8 equivale a 511.

Quando però si vuole intervenire nuovamente sulla variabile '**ibase**', occorre ricordare che per il momento la base di numerazione è otto. Pertanto, volendo tornare alla base 10, bisogna trasformare prima il valore in ottale: 12_8.

```
ibase=12[ Invio ]
```

777[*Invio*]

```
777
```

Diversamente, scrivendo nuovamente ‘**ibase=10**’ non si cambierebbe la base di numerazione, perché quel numero andrebbe inteso in ottale.

Esiste anche una convenzione, per cui i valori numerici espressi con una sola cifra, vanno intesi correttamente, in modo indipendente dal valore della variabile ‘**ibase**’. Pertanto, ‘**9**’ vale sempre come se fosse scritto in base 10, dal momento che non ci possono essere ambiguità anche se la base di numerazione fosse più grande.

Le cifre che possono essere usate per comporre un numero sono i simboli da ‘**0**’ a ‘**9**’, con le lettere maiuscole da ‘**A**’ a ‘**F**’. In questo modo si possono rappresentare agevolmente numeri con basi di numerazione che vadano da 2 a 16, mentre per basi di numerazione superiori le cose si complicano. In pratica, si possono rappresentare basi superiori scrivendo il risultato a cifre separate, dove ogni cifra è espressa come un numero in base 10. Per esempio, la stringa «100», esprimente un numero in base 20, verrebbe rappresentato come ‘**01 00 00**’.[1] L’esempio seguente mostra in che modo arrivare a questo risultato, tenendo in considerazione il fatto che la variabile ‘**ibase**’ contenga inizialmente il valore 10.

obase=20[*Invio*]

400[*Invio*]

```
01 00 00
```

In base al principio per il quale una cifra numerica singola viene interpretata in modo non ambiguo, indipendentemente dalla base di numerazione stabilita in ingresso con la variabile ‘**ibase**’, si può tornare facilmente a un inserimento di valori in base 10, sfruttando la cifra ‘**A**’, il cui valore è sempre pari a 10:

ibase=A[*Invio*]

311.2 Approssimazione

La variabile ‘**scale**’ definisce la quantità di cifre decimali da prendere in considerazione, quando il contesto non esprime già questo valore. In altri termini, una moltiplicazione definisce già la quantità di cifre decimali da considerare:

10*2.45[*Invio*]

```
24.50
```

Al contrario, nel caso della divisione è necessario stabilire subito la quantità di decimali da considerare:

scale=4[*Invio*]

10/3[*Invio*]

```
3.3333
```

Generalmente, all’avvio dell’interprete, il valore della variabile ‘**scale**’ è pari a zero, avendo così un’approssimazione predefinita alla parte intera.

[1] uno, zero, zero in base 20, corrisponde a 400.

311.3 Linguaggio di programmazione

Il linguaggio di BC ha una vaga somiglianza con il C. In generale, le righe vuote e quelle bianche vengono ignorate, così come il testo circoscritto tra '**/***' e '***/**' (proprio come nel C).

> Alcune realizzazioni di BC prevedono anche l'uso del simbolo '**#**' come commento, allo scopo di poter realizzare facilmente degli script, iniziando con '**#!/usr/bin/bc**', ma si tratta di un'estensione che non fa parte dello standard POSIX.

Le istruzioni del linguaggio BC terminano normalmente alla fine della riga, ma è possibile usare anche il punto e virgola ('**;**') se si preferisce, oppure se si vogliono indicare più istruzioni assieme sulla stessa riga. La continuazione di un'istruzione in una riga successiva si ottiene mettendo una barra obliqua inversa ('\') alla fine, esattamente prima del codice di interruzione di riga.

istruzione [**;**]

istruzione \
continuazione_istruzione

Si possono definire dei raggruppamenti di istruzioni, racchiudendoli tra parentesi graffe ('**{ }**'). Anche in questo caso le istruzioni possono essere separate attraverso interruzioni di riga, oppure con il punto e virgola.

{*istruzione*
istruzione
istruzione }

{*istruzione* **;** *istruzione* **;** *istruzione* }

Il linguaggio consente la dichiarazione di variabili e di funzioni, che possono avere un nome composto esclusivamente da una lettera minuscola. Alcune realizzazioni di BC consentono l'uso di nomi più articolati, ma si tratta di estensioni non compatibili con le specifiche POSIX.

> Il linguaggio BC non prevede una funzione principale, come avviene invece in C. Infatti, si tratta di un linguaggio interpretato dove tutto viene eseguito appena possibile; anche le funzioni esistono appena dichiarate e possono essere sostituite da una dichiarazione successiva che utilizza lo stesso nome

Esistono solo due tipi di dati: le stringhe delimitate e i valori numerici (con la quantità stabilita di cifre dopo la virgola), dove la separazione tra parte intera e parte decimale si indica esclusivamente con un punto ('**.**'). Tuttavia, alle variabili si possono assegnare solo numeri, così come le funzioni possono restituire solo valori numerici.

311.3.1 Variabili semplici e array

La dichiarazione di una variabile avviene in modo molto semplice, con l'assegnamento di un valore numerico, come nell'esempio seguente:

```
x=123.456
```

Nello stesso modo si possono dichiarare degli array a una sola dimensione, indicando un indice tra parentesi quadre, come nell'esempio seguente, dove in particolare l'indice è espresso da un'espressione:

```
x[1+2]=234.567
```

Gli array non devono essere dimensionati e possono usare la quantità massima di elementi disponibili in base alla realizzazione di BC. Il primo elemento si raggiunge con l'indice zero e gli elementi successivi sono numeri interi positivi. Se si fa riferimento a un elemento dell'array che non è ancora stato assegnato, si ottiene il valore zero.

Per fare riferimento a un array nel suo complesso, si indica il nome, seguito dalle parentesi quadre, aperte e chiuse, senza contenere alcun indice: '*x* **[]**'.

311.3.2 Funzioni

La dichiarazione di una funzione ha una forma precisa, dove in questo caso *x* rappresenta il nome della stessa:

```
define x( [parametro [, parametro ]]… ) {
    [auto variabile_automatica [, variabile_automatica ]… ]
    [istruzione ]
    …
    …
    [return [( varlore_restituito )]]
}
```

Si osservi in particolare l'uso delle parentesi graffe per delimitare il corpo della funzione: è indispensabile che la parentesi graffa aperta si trovi sulla stessa riga iniziale della dichiarazione della funzione, con i parametri relativi.

I parametri della funzione possono essere nomi di variabili normali, oppure nomi di array senza un indice tra le parentesi quadre.

I parametri che appaiono tra parentesi tonde, equivalgono alla dichiarazione implicita di variabili locali, definite di tipo *automatico*, contenenti il valore trasmesso al momento della chiamata.

Oltre alle variabili che compongono l'elenco dei parametri della funzione, si possono dichiarare altre variabili automatiche nel modo seguente, nella riga immediatamente successiva alla parentesi graffa aperta:

```
[auto variabile_automatica [, variabile_automatica ]… ]
```

Si può usare una sola istruzione '**auto**', nella quale vanno elencate tutte le variabili automatiche, compresi gli array, nella forma '*x* **[]**'.

Una funzione restituisce sempre un valore numerico, anche se non viene utilizzata esplicitamente l'istruzione '**return**'; in tal caso, si tratta sempre di zero.

> Se il valore restituito dalla funzione non viene usato nella chiamata per un assegnamento, questo viene visualizzato, anche se ciò non fosse desiderabile. Per evitare questo inconveniente, è possibile assegnare a una variabile fittizia il valore restituito dalla funzione.

> Anche se è possibile fornire un array come parametro in una chiamata di funzione, l'istruzione '**return**' non può restituire un array.

La chiamata di una funzione avviene nel modo seguente; anche in questo caso *x* rappresenta il nome della funzione chiamata:

```
x( [parametro [, parametro ]]… )
```

I parametri possono essere variabili oppure valori costanti. Nel primo caso, se la funziona cambia il contenuto delle variabili corrispondenti, tali modifiche non si riperquotono nelle variabili usate nella chiamata.

Le funzioni possono anche non avere parametri; in quei casi si indicano le parentesi tonde senza alcun contenuto, sia nella dichiarazione, sia nella chiamata.

L'esempio seguente, molto semplice, mostra la dichiarazione di una funzione che esegue la moltiplicazione:

```
define m (x, y) {
    auto z
    z=x*y
    return (z)
}
```

Se questa funzione venisse salvata nel file '`moltiplica`', si potrebbe usare BC nel modo seguente:

`$ bc moltiplica`[*Invio*]

`m (7, 2)`[*Invio*]

```
14
```

[*Ctrl+d*]

La parola chiave '**`return`**', può essere usata senza l'indicazione del valore da restituire e quindi senza le parentesi tonde. In tal caso viene restituito il valore zero.

311.3.3 Emissione delle informazioni

BC prevede poche funzioni predefinite (interne), ma non mette a disposizione una funzione per l'emissione di stringhe. Se necessario, una costante stringa viene visualizzata semplicemente indicandola come un'istruzione, con un piccolo accorgimento.

Un'espressione che si traduce in un numero, porta alla visualizzazione del risultato, seguito da un codice di interruzione di riga; pertanto,

```
4567*3456
```

seguito da [*Invio*], genera il risultato 15783552, che viene mostrato e il cursore viene quindi portato sulla riga successiva:

```
4567*3456
15783552
_
```

Al contrario, la visualizzazione di una stringa non fa avanzare alla riga successiva, permettendo l'aggiunta di altre stringhe e di un solo valore numerico finale. Infatti,

`"ciao " ; "amore " ; "bello!"`[*Invio*]

Si traduce in

```
ciao amore bello!_
```

con il cursore alla destra del punto esclamativo. Aggiungendo un numero, la visualizzazione sulla riga termina:

`"Anni: " ; 35`[*Invio*]

```
Anni: 35
_
```

Il risultato delle espressioni viene visualizzato se questo non viene catturato da un assegnamento a una variabile. Pertanto:

7*5[*Invio*]

35

Invece,

a=7*5[*Invio*]

non visualizza alcunché.

Tuttavia, è possibile mostrare il risultato di un'espressione il cui risultato viene assegnato a una variabile, racchiudendola all'interno di parentesi tonde. Pertanto:

(a=7*5)[*Invio*]

35

311.3.4 Espressioni

Gli operatori che intervengono su valori numerici sono elencati nella tabella 311.1. Esiste tuttavia un chiarimento da fare sull'espressione '*op1***%***op2*', che non si comporta secondo lo standard comune. Infatti, solo quando '**scale**' contiene il valore zero, il risultato è il resto della divisione intera; diversamente, si ottiene il resto della divisione, tolto il risultato ottenuto in base alla quantità di cifre decimali stabilito dalla variabile '**scale**'. Per esempio, se '**scale**' contiene il valore cinque, 10%3 genera il risultato 0,00001. Infatti, potendo gestire cinque cifre decimali, 10%3 dà il risultato 3,33333, per cui, il resto della divisione rimane solo 0,00001: 3,33333*3+0,00001=10.

Tabella 311.1. Elenco degli operatori aritmetici e di quelli di assegnamento relativi a valori numerici.

Operatore e operandi	Descrizione
++*op*	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
op++	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
--*op*	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
op--	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
+*op*	Non ha alcun effetto.
-*op*	Inverte il segno dell'operando.
op1 + *op2*	Somma i due operandi.
op1 - *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Modulo: il resto della divisione tra il primo e il secondo operando.
op1 ^ *op2*	Esponente: il primo operando elevato alla potenza del secondo.
x = *valore*	Assegna alla variabile il valore alla destra.
(*espressione*)	Le parentesi tonde richiedono la precedenza nella valutazione dell'espressione.
op1 += *op2*	*op1* = *op1* + *op2*
op1 -= *op2*	*op1* = *op1* - *op2*
op1 *= *op2*	*op1* = *op1* * *op2*
op1 /= *op2*	*op1* = *op1* / *op2*
op1 %= *op2*	*op1* = *op1* % *op2*

Alcune realizzazioni tradizionali di BC, non più standard secondo POSIX, consentono l'uso di operatori simili al tipo '*op***=**', descritti nella tabella, ma invertiti nell'ordine: '**=***op*'. Ciò crea un problema nella valutazione di alcuni tipi di espressione; per esempio, '**a=-1**' può significare l'assegnamento del valore -1 alla variabile '**a**', oppure l'assegnamento di '**a-1**'. Per evitare ambiguità in queste condizioni, conviene usare le parentesi: '**a=(-1)**'.

Tabella 311.2. Elenco degli operatori di assegnamento obsoleti che qualche realizzazione di BC potrebbe usare ancora.

Operatore e operandi	Descrizione
op1 =+ *op2*	*op1* = *op1* + *op2*
op1 =- *op2*	*op1* = *op1* - *op2*
op1 =* *op2*	*op1* = *op1* * *op2*
op1 =/ *op2*	*op1* = *op1* / *op2*
op1 =% *op2*	*op1* = *op1* % *op2*

Gli operatori di confronto determinano la relazione tra due operandi e possono essere utilizzati esclusivamente in alcuni contesti precisi. Vengono elencati gli operatori disponibili nella tabella 311.3.

Tabella 311.3. Elenco degli operatori di confronto. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 == *op2*	*Vero* se gli operandi si equivalgono.
op1 != *op2*	*Vero* se gli operandi sono differenti.
op1 < *op2*	*Vero* se il primo operando è minore del secondo.
op1 > *op2*	*Vero* se il primo operando è maggiore del secondo.
op1 <= *op2*	*Vero* se il primo operando è minore o uguale al secondo.
op1 >= *op2*	*Vero* se il primo operando è maggiore o uguale al secondo.

Lo standard POSIX stabilisce che queste espressioni possono apparire solo come condizione delle istruzioni '**`if`**', '**`while`**' e '**`for`**'; inoltre, è esclusa la possibilità di comporre espressioni più complesse con l'uso di operatori booleani.

311.3.5 Funzioni standard

BC predispone poche funzioni standard, che si distinguono in particolare per la lunghezza del loro nome. Queste sono riepilogate in breve nella tabella 311.4.

Tabella 311.4. Funzioni interne.

Funzione	Valore restituito
length (*espressione*)	Quantità di cifre significative dell'espressione.
scale (*espressione*)	Quantità di cifre decimali dell'espressione.
sqrt (*espressione*)	Radice quadrata dell'espressione.

Per quanto riguarda il caso particolare di '**`scale()`**', si fa riferimento al numero di decimali che genera l'espressione, in base al contesto. Per esempio, se il valore della variabile '**`scale`**' è zero, qualunque divisione darà soltanto un risultato intero, per cui '**`scale()`**' restituirà sempre solo zero.

Oltre a queste funzioni, è possibile chiedere a BC di mettere a disposizione alcune funzioni da una libreria standard. Si tratta di quelle elencate nella tabella 311.5.

Tabella 311.5. Funzioni della libreria standard.

Funzione	Valore restituito
s (*x*)	Seno.
c (*x*)	Coseno.
a (*x*)	Arcotangente.
l (*x*)	Logaritmo naturale.
e (*x*)	Funzione esponenziale: *e* elevato alla *x*.
j (*n*, *x*)	Funzione di Bessel.

311.3.6 Strutture di controllo di flusso

Il linguaggio BC gestisce le strutture di controllo di flusso principali, anche se con qualche limitazione. È disponibile una struttura condizionale semplificata (senza l'analisi di un'alternativa), il ciclo iterativo e il ciclo enumerativo:

```
if (condizione) istruzione
```

```
while (condizione) istruzione
```

```
for (espressione1; espressione2; espressione3) istruzione
```

Come nel linguaggio C, dal momento che si possono raggruppare le istruzioni in blocchi racchiusi tra parentesi graffe, in pratica si utilizzano queste strutture nel modo seguente:

```
if (condizione) {
    istruzione
    ...
}
```

```
while (condizione) {
    istruzione
    ...
}
```

```
for (espressione1; espressione2; espressione3) {
    istruzione
    ...
}
```

Naturalmente, le tre espressioni tra parentesi del ciclo enumerativo vanno intese nel modo comune. Per esempio, ciò che appare di seguito serve a mostrare 10 «x», attraverso il conteggio di una variabile.

```
for (i = 0; i < 10; i++) {
    "x"
}
```

Nell'ambito dei cicli, è possibile usare l'istruzione '**break**' per interrompere il ciclo con un'uscita forzata.

311.4 Utilizzo di BC

L'interprete del linguaggio BC è l'eseguibile '**bc**', che si utilizza secondo la sintassi seguente:

```
bc [-l] [file_bc]...
```

L'interprete legge ed esegue tutti i file indicati come argomento della riga di comando; alla fine, legge lo standard input. L'interprete termina di funzionare quando il flusso dello standard input termina, oppure quando incontra l'istruzione '**quit**'.

In questo modo, un programma che si deve concludere deve contenere l'istruzione '**quit**', oppure deve essere fornito attraverso lo standard input.

L'opzione '**-l**' serve a ottenere da BC la disponibilità delle funzioni di libreria standard, elencate nella tabella 311.5; inoltre, la variabile '**scale**' viene impostata al valore 20, mentre in condizioni normali il suo valore predefinito è zero.

Lo standard POSIX non prevede l'uso del simbolo '**#**' come commento, per cui non è possibile realizzare degli script se non sfruttando delle estensioni di realizzazioni speciali. In pratica, ci possono essere realizzazioni di BC che consentono di scrivere programmi che iniziano in modo simile a quello seguente, eventualmente con l'aggiunta dell'opzione '**-l**', a cui poi si aggiungono i permessi di esecuzione, ma ciò non è possibile se si vogliono scrivere programmi standard (portabili).

```
#!/usr/bin/bc
```

311.5 BC nella realizzazione GNU

La realizzazione GNU di BC [2] consente l'uso di diverse estensioni rispetto allo standard POSIX; in particolare completa la struttura di controllo condizionale con l'alternativa '**else**', aggiunge l'istruzione '**print**' per una gestione migliore della visualizzazione di informazioni e consente l'uso di operatori booleani nelle espressioni logiche, che possono essere usate anche al di fuori del contesto restrittivo stabilito da POSIX. Tuttavia è possibile richiedere un funzionamento strettamente aderente allo standard POSIX, utilizzando l'opzione '**-s**', oppure creando la variabile di ambiente '**POSIXLY_CORRECT**'.

L'eseguibile '**bc**' consente l'uso di più opzioni della riga di comando, alcune delle quali vengono descritte brevemente nel seguito.

Alcune opzioni

`-l`	`--mathlib`

Richiede l'uso delle librerie matematiche standard, impostando la variabile '**scale**' al valore 20.

`-w`	`--warn`

Segnala l'uso di estensioni allo standard POSIX.

`-s`	`--standard`

Restringe il funzionamento allo standard POSIX.

311.6 Riferimenti

- *IEEE P1003.2 Draft 11.2*, September 1991, *bc - Arbitrary-precision arithmetic language*
  <http://www.funet.fi/pub/doc/posix/p1003.2/d11.2/4.3>

[2] **GNU BC** GNU GPL

BC: esempi di programmazione

Questo capitolo raccoglie solo alcuni esempi di programmazione, in parte già descritti in altri capitoli. Per eseguire questi esempi basta usare il comando seguente, dove '`prova.b`' rappresenta il nome del file da eseguire:

```
$ bc prova.b
```

Si vuole evitare l'uso di estensioni al linguaggio BC, per cui i programmi non vengono mostrati come script; inoltre manca la possibilità di controllare l'interazione con l'utilizzatore, quindi le funzioni devono essere richiamate manualmente e al termine si deve usare il comando '**`quit`**', oppure si conclude il flusso dello standard input con la combinazione [*Ctrl+d*].

Negli esempi non si fa uso delle librerie standard, pertanto i nomi relativi possono essere riutilizzati.

Le espressioni vengono scritte in modo da evitare la visualizzazione non desiderata. Per esempio, invece di '**`i++`**', si preferisce usare la forma '**`i=(i+1)`**', quando possibile.

> Bisogna ricordare che se non si assegna il risultato generato da una funzione, questo viene visualizzato. La variabile '**`t`**' è stata usata negli esempi per raccogliere questo risultato quando non desiderato.

312.1 Problemi elementari di programmazione 3532

312.1.1 Somma tra due numeri positivi 3532
312.1.2 Moltiplicazione di due numeri positivi attraverso la somma 3532
312.1.3 Divisione intera tra due numeri positivi 3533
312.1.4 Elevamento a potenza 3533
312.1.5 Radice quadrata 3534
312.1.6 Fattoriale 3534
312.1.7 Massimo comune divisore 3535
312.1.8 Numero primo 3535

312.2 Scansione di array 3536

312.2.1 Ricerca sequenziale 3536
312.2.2 Ricerca binaria 3537

312.3 Algoritmi tradizionali 3537

312.3.1 Bubblesort 3537
312.3.2 Torre di Hanoi 3538
312.3.3 Quicksort 3539
312.3.4 Permutazioni 3540

312.1 Problemi elementari di programmazione

In questa sezione vengono mostrati alcuni algoritmi elementari portati in BC. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

312.1.1 Somma tra due numeri positivi

Il problema della somma tra due numeri positivi, attraverso l'incremento unitario, è stato descritto nella sezione 282.2.1.

```
/*
    somma.b
    Somma esclusivamente valori positivi.
*/

define s (x, y) {
    auto z, i
    z=x
    for (i=1; i<=y; i++) {
        z=(z+1)
    }
    return (z)
}

"Per calcolare la somma, si utilizzi la funzione s (x, y): "
```

In alternativa si può tradurre il ciclo '**for**' in un ciclo '**while**'.

```
define s (x, y) {
    auto z, i
    z=x
    i=1
    while (i<=y) {
        z=(z+1)
        i=(i+1)
    }
    return (z)
}
```

312.1.2 Moltiplicazione di due numeri positivi attraverso la somma

Il problema della moltiplicazione tra due numeri positivi, attraverso la somma, è stato descritto nella sezione 282.2.2.

```
/*
    moltiplica.b
*/

define m (x, y) {
    auto z, i
    z=0
    for (i=1; i<=y; i++) {
        z=(z+x)
    }
    return (z)
}

"Per calcolare la moltiplicazione, si utilizzi la funzione m (x, y): "
```

In alternativa si può tradurre il ciclo '**for**' in un ciclo '**while**'.

```
define m (x, y) {
    auto z, i
```

```
    z=0
    i=1
    while (i<=y) {
        z=(z+x)
        i=(i+1)
    }
    return (z)
}
```

312.1.3 Divisione intera tra due numeri positivi

Il problema della divisione tra due numeri positivi, attraverso la sottrazione, è stato descritto nella sezione 282.2.3.

```
/*
    dividi.b
    Divide esclusivamente valori positivi.
*/

define d (x, y) {
    auto z, i
    z=0
    i=x
    while (i>=y) {
        i=(i-y)
        z=(z+1)
    }
    return (z)
}

"Per calcolare la divisione intera, si utilizzi la funzione d (x, y): "
```

312.1.4 Elevamento a potenza

Il problema dell'elevamento a potenza tra due numeri positivi, attraverso la moltiplicazione, è stato descritto nella sezione 282.2.4.

```
/*
    exp.b
*/

define x (x, y) {
    auto z, i
    z=1
    for (i=1; i<=y; i++) {
        z=(z*x)
    }
    return (z)
}

"Per calcolare l'elevamento a potenza, si utilizzi la funzione x (x, y): "
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**'.

```
define x (x, y) {
    auto z, i
    z=1
    i=1
    while (i<=y) {
        z=(z*x)
        i=(i+1)
    }
    return (z)
}
```

È possibile usare anche un algoritmo ricorsivo.

```
define x (x, y) {
    if (x==0) {
        return (0)
    }
    if (y==0) {
        return (1)
    }
    return (x * x (x, y-1))
}
```

312.1.5 Radice quadrata

Il problema della radice quadrata è stato descritto nella sezione 282.2.5.

```
/*
    radice.b
*/

define r (x) {
    auto z, y
    z=0
    y=0
    while (1) {
        y=(z*z)
        if (y>x) {
            /* È stato superato il valore massimo. */
            z=(z-1)
            return (z)
        }
        z=(z+1)
    }
}

"Per calcolare la radice quadrata, si utilizzi la funzione r (x): "
```

312.1.6 Fattoriale

Il problema del fattoriale è stato descritto nella sezione 282.2.6.

```
/*
    fatt.b
*/

define f (x) {
    auto i
    i=(x-1)
    while (i>0) {
        x=(x*i)
        i=(i-1)
    }
    return (x)
}

"Per calcolare il fattoriale, si utilizzi la funzione f (x): "
```

In alternativa, l’algoritmo si può tradurre in modo ricorsivo.

```
define f (x) {
    if (x>1) {
        return (x * f (x-1))
    }
    return (1)
}
```

312.1.7 Massimo comune divisore

Il problema del massimo comune divisore, tra due numeri positivi, è stato descritto nella sezione 282.2.7.

```
/*
    mcd.b
*/

define m (x, y) {
    auto n
    while (x!=y) {
        n=0
        if (x>y) {
            x=x-y
            n=1
        }
        if (n==0) {
            y=(y-x)
        }
    }
    return (x)
}

"Per calcolare il massimo comune divisore, "
"si utilizzi la funzione m (x, y): "
```

312.1.8 Numero primo

Il problema della determinazione se un numero sia primo o meno, è stato descritto nella sezione 282.2.8.

```
/*
    primo.b
*/

define p(x) {
    auto i, j
    i=2
    while (i<x) {
        scale=0
        j=(x/i)
        j=x-(j*i)
        if (j==0) {
            return (0)
        }
        i=(i+1)
    }
    return (1)
}

"Per verificare se un numero sia primo, si utilizzi la funzione p (x, y); "
"1 indica un numero primo, 0 indica un numero che non è primo. "
```

312.2 Scansione di array

In questa sezione vengono mostrati alcuni algoritmi, legati alla scansione degli array, portati in BC. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

Per usare questi programmi, mancando un sistema normale di interazione con l'utilizzatore, è necessario creare un array prima di utilizzare la funzione che svolge il lavoro di ricerca o di riordino. Per esempio, nel caso della funzione '**`r()`**' per la ricerca sequenziale:

```
$ bc ricercaseq.b[ Invio ]

Ricerca sequenziale: r (<lista>, , <elemento>, <inizio>, <fine>)

a[0]=3[ Invio ]

a[1]=10[ Invio ]

a[2]=33[ Invio ]

a[3]=56[ Invio ]

r (a[], 33, 0, 3)[ Invio ]

2

[ Ctrl+d ]
```

312.2.1 Ricerca sequenziale

Il problema della ricerca sequenziale all'interno di un array, è stato descritto nella sezione 282.3.1.

```
/*
    ricercaseq.b
*/

/* r (<lista>, <elemento>, <inizio>, <fine>) */
define r (l[], x, a, z) {
    auto i
    for (i=a; i<=z; i++) {
        if (x==l[i]) {
            return (i)
        }
    }
    /* La corrispondenza non è stata trovata. */
    return (-1)
}

"Ricerca sequenziale: r (<lista>, , <elemento>, <inizio>, <fine>) "
```

Esiste anche una soluzione ricorsiva che viene mostrata nella funzione seguente:

```
define r (l[], x, a, z) {
    if (a>z) {
        return (-1)
    }
    if (x==l[a]) {
        return (a)
    }
    return (r (l[], x, a+1, z))
}
```

312.2.2 Ricerca binaria

Il problema della ricerca binaria all'interno di un array, è stato descritto nella sezione 282.3.2.

```
/*
    ricercabin.b
*/

/* r (<lista>, <elemento>, <inizio>, <fine>) */
define r (l[], x, a, z) {
    auto m
    /* Determina l'elemento centrale. */
    scale=0
    m = ((a+z)/2)
    if (m<a) {
        /* Non restano elementi da controllare: l'elemento cercato non c'è. */
        return (-1)
    }
    if (x<l[m]) {
        /* Si ripete la ricerca nella parte inferiore. */
        return (r (l[], x, a, m-1))
    }
    if (x>l[m]) {
        /* Si ripete la ricerca nella parte superiore. */
        return (r (l[], x, m+1, z))
    }
    /* $m rappresenta l'indice dell'elemento cercato. */
    return (m)
}

"Ricerca binaria: r (<lista>, <elemento>, <inizio>, <fine>) "
```

312.3 Algoritmi tradizionali

In questa sezione vengono mostrati alcuni algoritmi tradizionali portati in BC. Per la spiegazione degli algoritmi, se non sono già conosciuti, occorre leggere quanto riportato nel capitolo 282.

Per consentire la visualizzazione del contenuto di un array è necessario predisporre una funzione apposita, che viene presentata qui, senza ripeterla nei vari esempi proposti (per evitare di visualizzare uno zero aggiuntivo, conviene assegnare il valore restituito dalla funzione stessa).

```
/* v (<lista>, <inizio>, <fine>) */
define v (l[], a, z) {
    auto j
    for (j=a; j<=z; j++) {
        (l[j])
    }
    return
}
```

312.3.1 Bubblesort

Il problema del Bubblesort è stato descritto nella sezione 282.4.1. Viene mostrato prima una soluzione iterativa e successivamente la funzione '**`bsort()`**' in versione ricorsiva.

```
/*
    bsort.b
*/

/* l[] è l'array da riordinare. */

/* b (<inizio>, <fine>) */
```

```
define b (a, z) {
    auto s, j, k

    /* Inizia il ciclo di scansione dell'array. */
    for (j=a; j<z; j++) {
        /*
            Scansione interna dell'array per collocare nella posizione j
            l'elemento giusto.
        */
        for (k=(j+1); k<=z; k++) {
            if (l[k]<l[j]) {
                /* Scambia i valori */
                s=l[k]
                l[k]=l[j]
                l[j]=s
            }
        }
    }
    return
}

"Bubblesort: l[]; t = b (<inizio>, <fine>)   "
"L'array da riordinare è l[]. "
```

Segue la funzione '**`bsort()`**' in versione ricorsiva.

```
define b (a, z) {
    auto s, k
    if (a<z) {
        /*
            Scansione interna dell'array per collocare nella posizione a
            l'elemento giusto.
        */
        for (k=(a+1); k<=z; k++) {
            if (l[k]<l[a]) {
                /* Scambia i valori */
                s=l[k]
                l[k]=l[a]
                l[a]=s
            }
        }
        b (a+1, z)
    }
    return
}
```

312.3.2 Torre di Hanoi

Il problema della torre di Hanoi è stato descritto nella sezione 282.4.2.

```
/*
    hanoi.b
*/

/* h (<n-anelli>, <piolo-iniziale>, <piolo-finale>) */
define h (n, i, f) {
    auto t
    if (n>0) {
        t = h (n-1, i, 6-i-f)
        "Muovi l'anello " ; n
        "dal piolo " ; i
        "al piolo " ; f
        t = h (n-1, 6-i-f, f);
    }
    return
}
```

```
"Torre di Hanoi: t = h (<n-anelli>, <piolo-iniziale>, <piolo-finale>) "
```

312.3.3 Quicksort

L'algoritmo del Quicksort è stato descritto nella sezione 282.4.3.

```
/*
    qsort.b
*/

/* l[] è l'array da riordinare. */

/* p (<inizio>, <fine>) */
define p (a, z) {
    auto s, i, c
    /* Si assume che a sia inferiore a z. */
    i=(a+1)
    c=z

    /* Inizia il ciclo di scansione dell'array. */
    while (1) {
        while (1) {
            /* Sposta i a destra. */
            if (l[i]>l[a]) {
                break
            }
            if (i>=c) {
              break
            }
            i=(i+1)
        }
        while (1) {
            /* Sposta c a sinistra. */
            if (l[c]<=l[a]) {
                break
            }
            c=(c-1)
        }
        if (c<=i) {
            /* È avvenuto l'incontro tra i e c. */
            break
        }
        /* Vengono scambiati i valori. */
        s=l[c]
        l[c]=l[i]
        l[i]=s

        i=(i+1)
        c=(c-1)
    }

    /*
        A questo punto l[a..z] è stata ripartita e c è la collocazione
        di l[a].
    */
    s=l[c]
    l[c]=l[a]
    l[a]=s

    /*
        A questo punto l[c] è un elemento (un valore) nella
        posizione giusta.
    */
    return (c)
}
```

```
/* q (<inizio>, <fine>) */
define q (a, z) {
    auto c
    if (z>a) {
        c = p (a, z)
        q (a, c-1)
        q (c+1, z)
    }
    return
}

"Quicksort: l[]  t = q (<inizio>, <fine>)  "
"Prima riempire l'array l[], poi chiamare la funzione q()."
```

312.3.4 Permutazioni

L'algoritmo ricorsivo delle permutazioni è stato descritto nella sezione 282.4.4.

```
/*
    permuta.b
*/

/* v (<lista>, <inizio>, <fine>) */
define v (l[], a, z) {
    auto j
    for (j=a; j<=z; j++) {
        (l[j])
    }
    return
}

/* p (<lista>, <inizio>, <fine>, <max_array>) */
define p (l[], a, z, d) {
    auto k
    auto t
    if ((z-a)>=1) {
        /*
            Inizia un ciclo di scambi tra l'ultimo elemento e uno degli
            altri contenuti nel segmento di array.
        */
        for (k=z; k>=a; k--) {
            /* Scambia i valori */
            s=l[k]
            l[k]=l[z]
            l[z]=s

            /*
                Esegue una chiamata ricorsiva per permutare un segmento
                più piccolo dell'array.
            */

            t = p (l[], a, z-1, d)

            /* Scambia i valori */
            s=l[k]
            l[k]=l[z]
            l[z]=s
        }
        return
    }
    /* Visualizza la situazione attuale dell'array. */
    " "
    t = v (l[],0,d)
    return
}
```

```
"Permutazioni: t = p (<lista>, <inizio>, <fine>, <max_array>)"
```

Parte lxi

Basic

313 Basic: introduzione 3544

313.1 Struttura fondamentale 3544
313.2 Interprete tradizionale 3545
313.3 Tipi di dati ed espressioni 3546
313.4 Primi esempi banali 3548
313.5 Strutture di controllo del flusso 3549
313.6 Input e output 3550

314 Basic: esempi di programmazione 3551

314.1 Somma tra due numeri positivi 3551
314.2 Moltiplicazione di due numeri positivi attraverso la somma 3551
314.3 Divisione intera tra due numeri positivi 3552
314.4 Elevamento a potenza 3552
314.5 Radice quadrata 3552
314.6 Fattoriale 3553
314.7 Ricerca sequenziale 3553

Basic: introduzione

Il Basic è un linguaggio di programmazione nato solo per scopi didattici, anche se ormai non si può più considerare tanto adatto neanche per questo. La semplicità di questo linguaggio fa sì che si trovino quasi sempre solo interpreti e non compilatori; in ogni caso, la natura stessa del linguaggio è tale per cui questo dovrebbe sempre essere solo interpretato.

313.1 Struttura fondamentale

Di linguaggi Basic ne esistono tanti tipi, anche con estensioni che vanno molto lontano rispetto all'impostazione originale, facendone in realtà un linguaggio completamente diverso. In questa descrizione, si vuole fare riferimento al Basic tradizionale, con tutte le sue limitazioni antiche. In questo senso, l'interprete Basic per GNU/Linux che più si avvicina a questo livello è Bywater BASIC. [1]

313.1.1 Numerazione delle righe

La caratteristica tipica di un programma Basic è quella di avere le righe numerate. Infatti, non gestendo procedure e funzioni, l'unico modo per accedere a una subroutine è quella di fare riferimento alla riga in cui questa inizia. In pratica, le istruzioni iniziano con un numero di riga, progressivo, seguito da almeno uno spazio; quindi continuano con l'istruzione vera e propria.

```
110 PRINT "ciao a tutti"
120 PRINT "come va?"
```

Si può intendere che questa dipendenza dalla numerazione delle righe costituisca poi un problema per il programmatore, perché il cambiamento di questa numerazione implica la perdita dei riferimenti alle subroutine.

313.1.2 Istruzioni

Le istruzioni Basic, oltre al fatto di iniziare con il numero di riga, non hanno altre caratteristiche particolari. Generalmente utilizzano una riga e non richiedono la conclusione finale con un qualche simbolo di interpunzione.

È interessante notare invece che i commenti vanno espressi con l'istruzione '**REM**', seguita da qualcosa che poi viene ignorato, e che le righe vuote non sono ammissibili in generale, anche se iniziano regolarmente con il numero di riga.

La natura del linguaggio Basic è tale per cui le istruzioni e i nomi delle variabili dovrebbero essere espressi sempre utilizzando le sole lettere maiuscole.

313.1.3 Esecuzione di un programma

L'esecuzione di un programma Basic dipende dal modo stabilito dall'interprete prescelto. L'interprete tradizionale obbliga a caricare il programma con il comando '**LOAD**' e ad avviarlo attraverso il comando '**RUN**'.

[1] **Bywater BASIC** GNU GPL

313.2 Interprete tradizionale

L'interprete Basic tradizionale è una sorta di shell che riconosce una serie di comandi interni, oltre alle istruzioni Basic vere e proprie. In pratica, attraverso l'invito di questa shell si possono eseguire singole istruzioni Basic, oppure comandi utili a gestire il file di un programma completo. Per esempio, avviando il Bywater BASIC, si ottiene quanto segue:

```
$ bwbasic[ Invio ]
bwBASIC:
```

In pratica, '**bwBASIC:**' rappresenta l'invito. L'esempio seguente mostra l'inserimento di alcune istruzioni Basic, allo scopo di eseguire la moltiplicazione 6*7.

```
bwBASIC: A=6[ Invio ]

bwBASIC: B=7[ Invio ]

bwBASIC: C=A*B[ Invio ]

bwBASIC: PRINT C[ Invio ]
42
```

313.2.1 Comandi tipici dell'interprete

L'interprete Basic tipico mette a disposizione alcuni comandi, che risultano essenziali per la gestione di un programma Basic.

- `LIST [riga_iniziale [-riga_finale]] [,…]`

  Elenca le righe del programma selezionate dagli intervalli indicati come argomento. Se non viene indicato alcun argomento, la visualizzazione viene fatta a partire dalla prima riga; se viene indicata solo la riga iniziale, la visualizzazione riguarda esclusivamente quella riga. L'esempio seguente serve a visualizzare la riga 100 e poi l'intervallo da 150 a 200.

  ```
  LIST 100, 150-200
  ```

- `RUN [riga_iniziale]`

  Il comando '**RUN**' viene usato normalmente senza argomenti, per avviare il programma caricato nell'interprete. Se si aggiunge il numero di una riga, quel punto verrà utilizzato per iniziare l'interpretazione ed esecuzione del programma.

- `NEW`

  Cancella il programma che eventualmente fosse caricato nell'interprete.

- `LOAD file`

  Carica il programma indicato dal nome del file posto come argomento. Se esisteva precedentemente un programma in memoria, quello viene eliminato. Solitamente, il nome del file deve essere indicato delimitandolo tra apici doppi. È probabile che l'interprete aggiunga un'estensione predefinita od obbligatoria.

- `SAVE file`

  Salva il programma con il nome specificato come argomento. Solitamente, il nome del file deve essere indicato delimitandolo tra apici doppi. È probabile che l'interprete aggiunga un'estensione predefinita od obbligatoria.

- `DEL` *riga_iniziale* [- *riga_finale*] [, …]

  Elimina le righe indicate dall'argomento. Può trattarsi di una sola riga, o di un intervallo, o di una serie di intervalli.

- `RENUM` [*riga_iniziale* [, *incremento*]]

  Rinumera le righe del programma, aggiornando i riferimenti alle subroutine. È possibile indicare il numero iniziale e anche l'incremento. Di solito, se non viene specificato alcun argomento, la riga iniziale ha il numero 10 e l'incremento è sempre di 10.

- `BYE` | `QUIT`

  Termina il funzionamento dell'interprete Basic.

L'inserimento delle righe di programma attraverso l'interprete Basic, avviene iniziando le istruzioni con il numero di riga in cui queste devono essere collocate. Ciò permette così di inserire righe aggiuntive anche all'interno del programma. Se si utilizzano numeri di righe già esistenti, queste vengono sostituite.

Quando un'istruzione Basic viene inserita senza il numero iniziale, questa viene eseguita immediatamente.

313.3 Tipi di dati ed espressioni

I tipi di dati gestibili in Basic sono generalmente solo i numeri reali (numeri a virgola mobile con approssimazione che varia a seconda dell'interprete) e le stringhe.

I numeri vengono indicati senza l'uso di delimitatori; se necessario, è possibile rappresentare valori decimali con l'uso del punto di separazione; inoltre è generalmente ammissibile la notazione esponenziale. L'esempio seguente mostra due modi di rappresentare lo stesso numero.

```
123.456
1.23456E+2
```

Le stringhe si rappresentano delimitandole attraverso apici doppi (possono essere ammessi anche gli apici singoli, ma questo dipende dall'interprete) e sono soggette a un limite di dimensione che dipende dall'interprete (spesso si tratta di soli 255 caratteri).

Le variabili sono distinte in base al fatto che servano a contenere numeri o stringhe. Per la precisione, le variabili che contengono stringhe, hanno un nome che termina con il simbolo dollaro ('**$**'). I nomi delle variabili, a parte l'eventuale aggiunta del dollaro per le stringhe, sono soggetti a regole differenti a seconda dell'interprete; in particolare occorre fare attenzione al fatto che l'interprete potrebbe distinguere tra maiuscole e minuscole. In origine, si poteva utilizzare una sola lettera alfabetica!

L'assegnamento di una variabile avviene attraverso l'operatore '**=**', secondo la sintassi seguente:

[`LET`] *variabile=valore*

L'uso esplicito dell'istruzione '**LET**' è facoltativo.

313.3.1 Espressioni numeriche

Gli operatori tipici che intervengono su valori numerici, restituendo valori numerici, sono elencati nella tabella 313.1.

Tabella 313.1. Elenco degli operatori utilizzabili in presenza di valori numerici, all'interno di espressioni numeriche. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
var = *valore*	Assegna alla variabile il valore alla destra.
- *op1*	Inverte il segno dell'operando.
op1 + *op2*	Somma i due operandi.
op1 - *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 MOD *op2*	Modulo: il resto della divisione tra il primo e il secondo operando.
op1 ^ *op2*	Eleva il primo operando alla potenza del secondo.
SQRT *op1*	Calcola la radice quadrata dell'operando.
SIN *op1*	Calcola il seno dell'operando.
COS *op1*	Calcola il coseno dell'operando.
TAN *op1*	Calcola la tangente dell'operando.
ARCTAN *op1*	Calcola l'arcotangente dell'operando.
LOG *op1*	Calcola il logaritmo naturale dell'operando.
ABS *op1*	Calcola il valore assoluto dell'operando.

Le parentesi tonde possono essere utilizzate per indicare esplicitamente l'ordine dell'elaborazione delle espressioni.

313.3.2 Espressioni stringa

L'unico tipo di espressione che restituisce una stringa a partire da stringhe, è il concatenamento che si ottiene con l'operatore '**+**'.

```
stringa_1+stringa_2
```

313.3.3 Espressioni logiche

Le espressioni logiche si possono realizzare a partire da dati numerici, da dati stringa e dal risultato di altre espressioni logiche. La tabella 313.2 mostra gli operatori fondamentali.

Tabella 313.2. Elenco degli operatori utilizzabili nelle espressioni logiche. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 = *op2*	I due numeri, o le due stringhe sono uguali.
op1 < *op2*	Il primo operando è minore del secondo.
op1 > *op2*	Il primo operando è maggiore del secondo.
op1 <= *op2*	Il primo operando è minore o uguale al secondo.
op1 >= *op2*	Il primo operando è maggiore o uguale al secondo.
op1 <> *op2*	I due operandi sono diversi.
cond1 AND *cond2*	Le due condizioni sono entrambe vere.
cond1 OR *cond2*	Almeno una delle due condizioni è vera.
NOT *cond1*	Inverte il risultato logico della condizione.

313.3.4 Espressioni miste

Alcuni operatori utilizzano valori di tipo diverso dal tipo di dati che restituiscono. La tabella 313.3 mostra alcuni di questi.

Tabella 313.3. Elenco di altri operatori.

Operatore e operandi	Descrizione
VAL *stringa*	Valuta la stringa trattandola come un'espressione numerica.
LEN *stringa*	Restituisce la lunghezza della stringa in caratteri.
STR$ *numero*	Restituisce una stringa contenente il numero indicato come argomento.

313.3.5 Array

Gli array in Basic possono essere a una o più dimensioni, a seconda dell'interprete. In ogni caso, dovrebbero essere distinti in base al contenuto: solo numeri o solo stringhe. L'indice del primo elemento dovrebbe essere zero. La dichiarazione avviene nel modo seguente:

`DIM` *nome* `(` *dimensione_1* [`,` *dimensione_2*]... `)`

`DIM` *nome*`$(` *dimensione_1* [`,` *dimensione_2*]... `)`

Nel primo caso si tratta di un array con elementi numerici, nel secondo si tratta di un array con elementi stringa.

313.4 Primi esempi banali

L'esempio seguente è il più banale, emette semplicemente la stringa '**`"Ciao Mondo!"`**' attraverso lo standard output.

```
10 print "Ciao Mondo!"
```

Per eseguire il programma basta utilizzare il comando '**`RUN`**'.

`RUN`[*Invio*]

```
Ciao Mondo!
```

L'esempio seguente genera lo stesso risultato di quello precedente, ma con l'uso di variabili.

```
10 A$ = "Ciao"
20 B$ = "Mondo"
30 PRINT A$; " "; B$
```

L'esempio seguente genera lo stesso risultato di quello precedente, ma con l'uso del concatenamento di stringa.

```
10 A$ = "Ciao"
20 B$ = "Mondo"
30 PRINT A$+" "+B$
```

L'esempio seguente mostra l'uso di una costante e di una variabile numerica.

```
10 A$ = "Ciao"
20 B$ = "Mondo"
30 N = 1000
40 PRINT N; "volte "; A$; " "; B$
```

Il risultato che si ottiene dovrebbe essere il seguente:

```
1000 volte Ciao Mondo!
```

313.5 Strutture di controllo del flusso

Il Basic è un linguaggio di programmazione molto povero dal punto di vista delle strutture di controllo. In modo particolare sono assenti funzioni e procedure. Per fare riferimenti a porzioni di codice occorre sempre indicare un numero di riga, attraverso le istruzioni '**GOTO**' o '**GOSUB**'.

313.5.1 GOTO

```
GOTO riga
```

Si tratta dell'istruzione di salto incondizionato e senza ritorno. In pratica, l'esecuzione del programma prosegue dalla riga indicata come argomento, perdendo ogni riferimento al punto di origine.

313.5.2 GOSUB

```
GOSUB riga
```

Si tratta dell'istruzione di salto incondizionato con ritorno. L'esecuzione del programma prosegue dalla riga indicata come argomento e, quando poi viene incontrata l'istruzione '**RETURN**', il programma riprende dalla riga successiva a quella in cui era avvenuta la chiamata. Questo è l'unico modo offerto dal Basic tradizionale per la realizzazione di subroutine

L'esempio seguente mostra un programma completo che visualizza il messaggio '**"Ciao"**' e poi il messaggio '**"Mondo"**'.

```
10 GOTO 50
20 A$ = "Ciao"
30 PRINT A$
40 RETURN
50 GOSUB 20
60 B$ = "Mondo"
70 PRINT B$
```

313.5.3 IF

```
IF condizione THEN istruzione [ELSE istruzione]
```

Se la condizione si verifica, viene eseguita l'istruzione posta dopo la parola chiave '**THEN**', altrimenti, se esiste, quella posta dopo la parola chiave '**ELSE**'. La situazione è tale per cui le istruzioni condizionate saranno prevalentemente '**GOTO**' e '**GOSUB**'.

L'esempio seguente emette la stringa '**"Ottimo"**' se la variabile '**N**' contiene un valore superiore a 100; altrimenti esegue la subroutine che inizia a partire dalla riga 50.

```
150 IF N > 100 THEN PRINT "Ottimo" ELSE GOSUB 50
```

313.5.4 FOR

```
FOR variabile_num = inizio TO fine [STEP incremento]
istruzioni
...
NEXT
```

Esegue le istruzioni e ogni volta incrementa la variabile numerica indicata, assegnandole inizialmente il valore posto dopo il simbolo '**=**'. Il blocco di istruzioni viene eseguito fino a quando la variabile raggiunge il valore finale stabilito; l'incremento è unitario, a meno che sia stato indicato diversamente attraverso l'argomento della parola chiave '**STEP**'.

313.5.5 END, STOP

La conclusione, o l'interruzione del programma può essere indicata esplicitamente utilizzando l'istruzione '**END**' oppure l'istruzione '**STOP**'. La prima corrisponde all'interruzione dovuta a una conclusione normale, la seconda serve a generare un messaggio di errore e si presta per l'interruzione del programma in presenza di situazioni anomale.

313.6 Input e output

L'input e l'output del Basic tradizionale è molto povero, riguardando prevalentemente l'acquisizione di dati da tastiera e l'emissione di testo sullo schermo.

313.6.1 PRINT

```
PRINT operando [{, | ;}...]
```

L'istruzione '**PRINT**' permette di emettere sullo schermo una stringa corrispondente agli operandi utilizzati come argomenti. Eventuali valori numerici vengono convertiti in stringhe automaticamente. Gli operandi possono essere elencati utilizzando la virgola o il punto e virgola. Gli esempi seguenti sono equivalenti.

```
10 PRINT 1234, "saluti"

10 PRINT 1234; "saluti"

10 A = 1234
20 PRINT A; "saluti"

10 A = 1234
20 M$ = "saluti"
30 PRINT A; M$
```

Se come operando si vuole utilizzare il risultato di un'espressione, di qualunque tipo, può essere necessario l'uso di parentesi tonde, come nell'esempio seguente, in cui si vuole emettere il risultato del coseno di zero.

```
10 PRINT ( COS 0 )
```

313.6.2 INPUT, ?

```
INPUT [invito ;] variabile [, variabile]...
```

```
? [invito ;] variabile [, variabile]...
```

Attraverso questa istruzione è possibile inserire un valore in una variabile, o una serie di valori in una serie di variabili. Se viene indicata la stringa dell'invito, questa viene visualizzata prima di attendere l'inserimento da parte dell'utente; altrimenti viene visualizzato semplicemente un punto interrogativo.

Se si indica un elenco di variabili, queste devono essere dello stesso tipo (tutte numeriche o tutte stringa) e il loro inserimento viene atteso in modo sequenziale da parte dell'utente.

L'esempio seguente rappresenta l'inserimento di una stringa senza invito e di una coppia di numeri con invito.

```
10 INPUT A$
20 INPUT "Inserisci la coppia di numeri "; X, Y
```

Basic: esempi di programmazione

In questo capitolo si raccolgono solo alcuni esempi molto semplici di programmazione in Basic. Infatti, questo linguaggio di programmazione non si presta per la rappresentazione di algoritmi complessi.

314.1 Somma tra due numeri positivi 3551
314.2 Moltiplicazione di due numeri positivi attraverso la somma 3551
314.3 Divisione intera tra due numeri positivi 3552
314.4 Elevamento a potenza 3552
314.5 Radice quadrata 3552
314.6 Fattoriale 3553
314.7 Ricerca sequenziale 3553

314.1 Somma tra due numeri positivi

Il problema della somma tra due numeri positivi, attraverso l'incremento unitario, è stato descritto nella sezione 282.2.1.

```
1000 REM ============================================================
1010 REM somma.bas
1020 REM Somma esclusivamente valori positivi.
1030 REM ============================================================
1040 REM
1050 INPUT "Inserisci il primo valore "; X
1060 INPUT "Inserisci il secondo valore "; Y
1070 LET Z = X
1080 FOR I = 1 TO Y
1090 LET Z = Z + 1
1100 NEXT
1110 PRINT X; "+"; Y; "="; Z
1120 END
1130 REM ============================================================
```

314.2 Moltiplicazione di due numeri positivi attraverso la somma

Il problema della moltiplicazione tra due numeri positivi, attraverso la somma, è stato descritto nella sezione 282.2.2.

```
1000 REM ============================================================
1010 REM moltiplica.bas
1020 REM Moltiplica esclusivamente valori positivi.
1030 REM ============================================================
1040 REM
1050 INPUT "Inserisci il primo valore "; X
1060 INPUT "Inserisci il secondo valore "; Y
1070 LET Z = 0
1080 FOR I = 1 TO Y
1090 LET Z = Z + X
1100 NEXT
1110 PRINT X; "*"; Y; "="; Z
1120 END
1130 REM ============================================================
```

314.3 Divisione intera tra due numeri positivi

Il problema della divisione tra due numeri positivi, attraverso la sottrazione, è stato descritto nella sezione 282.2.3.

```
1000 REM ============================================================
1010 REM dividi.bas
1020 REM Divide esclusivamente valori positivi.
1030 REM ============================================================
1040 REM
1050 INPUT "Inserisci il primo valore "; X
1060 INPUT "Inserisci il secondo valore "; Y
1070 LET Z = 0
1080 LET I = X
1090 IF I < Y THEN GOTO 1130
1100 LET I = I - Y
1110 LET Z = Z + 1
1120 GOTO 1090
1130 PRINT X; "/"; Y; "="; Z
1140 END
1150 REM ============================================================
```

314.4 Elevamento a potenza

Il problema dell'elevamento a potenza tra due numeri positivi, attraverso la moltiplicazione, è stato descritto nella sezione 282.2.4.

```
1000 REM ============================================================
1010 REM exp.bas
1020 REM Eleva a potenza.
1030 REM ============================================================
1040 REM
1050 INPUT "Inserisci il primo valore "; X
1060 INPUT "Inserisci il secondo valore "; Y
1070 LET Z = 1
1080 FOR I = 1 TO Y
1090 LET Z = Z * X
1100 NEXT
1110 PRINT X; "^"; Y; "="; Z
1120 END
1130 REM ============================================================
```

314.5 Radice quadrata

Il problema della radice quadrata è stato descritto nella sezione 282.2.5.

```
1000 REM ============================================================
1010 REM radice.bas
1020 REM Radice quadrata intera.
1030 REM ============================================================
1040 REM
1050 INPUT "Inserisci il valore "; X
1060 LET Z = 0
1070 LET T = 0
1080 REM Inizio del ciclo di calcolo
1090 LET T = Z * Z
1100 IF T > X THEN GOTO 1130
1110 LET Z = Z + 1
1120 GOTO 1080
1130 REM Riprende il flusso normale
1140 LET Z = Z - 1
1150 PRINT "radq("; X; ") ="; Z
1160 END
1170 REM ============================================================
```

314.6 Fattoriale

Il problema del fattoriale è stato descritto nella sezione 282.2.6.

```
1000 REM ============================================================
1010 REM fatt.bas
1020 REM Fattoriale.
1030 REM ============================================================
1040 REM
1050 INPUT "Inserisci il valore "; X
1060 LET Z = X
1070 FOR I = (X - 1) TO 1 STEP -1
1080 LET Z = Z * I
1090 NEXT
1100 PRINT "fatt("; X; ") ="; Z
1110 END
1120 REM ============================================================
```

314.7 Ricerca sequenziale

Il problema della ricerca sequenziale all'interno di un array, è stato descritto nella sezione 282.3.1.

```
1000 REM ============================================================
1010 REM ricercaseq.bas
1020 REM Ricerca sequenziale.
1030 REM ============================================================
1040 REM
1050 INPUT "Inserisci il numero di elementi "; N
1060 DIM A(N)
1070 FOR I = 0 TO N-1
1080 PRINT "A("; I; ") ="
1090 INPUT A(I)
1100 NEXT
1110 INPUT "Inserisci il valore da cercare "; X
1120 FOR I = 0 TO N-1
1130 IF X = A(I) THEN GOTO 1170
1140 NEXT
1160 GOTO 1190
1170 PRINT "L'elemento A("; I; ") contiene il valore "; X
1180 END
1190 PRINT "Il valore "; X; " non è stato trovato"
1200 END
1210 REM ============================================================
```

Parte lxii

Nazionalizzazione e localizzazione

315 Gettext: introduzione 3556

315.1 Principio di funzionamento 3556
315.2 Fasi di preparazione 3556
315.3 Abbinamento a un «pacchetto» 3558
315.4 Creazione e mantenimento dei file PO 3559
315.5 Gettext con i programmi Perl 3560

Indice analitico del volume 3563

Gettext: introduzione

Gettext [1] è un sistema che aiuta nella traduzione dei messaggi dei programmi e al loro mantenimento. Ci possono essere molti modi per realizzare un programma multilingua, ma Gettext rappresenta probabilmente il metodo più semplice in pratica che consente la traduzione successiva senza interferire con un eseguibile già pronto, purché predisposto per questo.

315.1 Principio di funzionamento

La logica di Gettext è molto semplice: il programma incorpora solo i messaggi in inglese; all'esterno si associano una serie di file, uno per ogni linguaggio disponibile, con le traduzioni corrispondenti. Non è necessario «codificare» i messaggi in qualche modo, perché la corrispondenza avviene in modo letterale, in base al testo originale.

```
msgid "%s: cannot create the temporary file %s\n"
msgstr "%s: non è possibile creare il file temporaneo %s\n"
```

L'esempio, che mostra un estratto ipotetico di un file PO di Gettext (*Portable object*), serve a comprendere il concetto: La stringa preceduta dalla parola chiave '**`msgid`**' (*message identity*) è quella di riferimento, che viene rimpiazzata automaticamente da quella sottostante, preceduta dalla parola chiave '**`msgstr`**'.

Le stringhe e le traduzioni di Gettext sono costanti, nel senso che '**`%s`**' viene preso come tale, mentre è il programma che lo sostituisce opportunamente. In questo senso, bisogna considerare che Gettext è nato per il linguaggio C, per essere usato in stringhe che siano argomento di funzioni come '**`printf()`**' e '**`sprintf()`**'.

315.2 Fasi di preparazione

La predisposizione di un programma per Gettext potrebbe essere fatta in modo più o meno automatico, attraverso strumenti specifici, oppure si può procedere in modo più semplice, anche se più oneroso dal punto di vista del tempo impiegato. Qui si intende mostrare questo modo più semplice per permettere al lettore di comprendere il concetto. La documentazione di Gettext è di per sé molto dettagliata.

Per prima cosa, il sorgente C deve essere predisposto attraverso l'inclusione di alcuni file di intestazione, quindi le stringhe vengono inglobate dalla funzione '**`gettext()`**'. Quello che segue è il classico programma che visualizza un messaggio ed esce; si suppone che si tratti del file '`ciao.c`':

```
#include <stdio.h>
int main ()
{
    printf ("Hello world\n");
}
```

Ecco come deve essere trasformato:

```
#include <stdio.h>
#include <libintl.h>
#include <locale.h>

#define PACKAGE "ciao"
#define LOCALEDIR "/var/tmp"
```

[1] **Gettext** GNU GPL

```
int main ()
{
    setlocale (LC_ALL, "");
    bindtextdomain (PACKAGE, LOCALEDIR);
    textdomain (PACKAGE);

    printf (gettext ("Hello world\n"));
}
```

Le funzioni '**bindtextdomain**' e '**textdomain**' utilizzano come argomenti delle macro (costanti manifeste), in modo da generalizzare il funzionamento e rendere esterna la definizione di queste componenti. A parte questi particolari, si nota che '**printf()**' non ha più come argomento la costante di prima, ma la funzione '**gettext()**'.

Il programma può essere compilato, anche se per adesso non c'è alcuna traduzione disponibile per lui.

```
$ cc -o ciao ciao.c
```

La fase successiva richiede la creazione di un file PO, attraverso l'aiuto del programma '**xgettext**':

```
$ xgettext ciao.c
```

Quello che si ottiene nella directory corrente è il file '`messages.po`', contenente esattamente il testo seguente:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR Free Software Foundation, Inc.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 2000-05-15 23:05+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: ENCODING\n"

#: ciao.c:18
msgid "Hello world\n"
msgstr ""
```

Questo file deve essere modificato, in particolare per ciò che riguarda le prime direttive, oltre che per aggiungere la traduzione della frase che viene visualizzata dal programma. Per esempio, così:

```
# Ciaomondo PO file.
# Copyright (C) 2000 Pinco Pallino
# Pinco Pallino <ppinco@dinkel.brot.dg>, 2000.
#
msgid ""
msgstr ""
"Project-Id-Version: ciao-0.1\n"
"POT-Creation-Date: 2000-05-15 23:05+0200\n"
"PO-Revision-Date: 2000-05-15 22:52+0200\n"
"Last-Translator: Pinco Pallino <ppinco@dinkel.brot.dg>\n"
"Language-Team: Italian <it@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=iso-8859-1\n"
"Content-Transfer-Encoding: 8bit\n"
```

```
#: ciao.c:18
msgid "Hello world\n"
msgstr "Ciao mondo\n"
```

Si osservi che è stato necessario togliere la riga che conteneva il commento speciale '**#, fuzzy**'.

Il file viene salvato con un nome appropriato; per esempio '`ciao.po`'. Quindi si passa alla sua compilazione, per ottenere il file '`ciao.mo`':

```
$ msgfmt -vvvv -o ciao.mo ciao.po
```

Dal momento che il file in questione contiene la traduzione in italiano del programma, deve essere collocato all'interno della gerarchia '`it/LC_MESSAGES/`', a sua volta a partire dalla directory dichiarata con la funzione '**bindtextdomain()**', cioè '`/var/tmp/it/LC_MESSAGES/`' secondo quando definito nel sorgente C.

A questo punto, dopo la collocazione appropriata del file compilato della traduzione, se la configurazione locale è corretta, lanciando l'eseguibile '**ciao**' si dovrebbe vedere il messaggio tradotto. Eventualmente si veda quando descritto nel capitolo 58 per quanto riguarda la configurazione della localizzazione.

315.3 Abbinamento a un «pacchetto»

Perché Gettext sappia qual è il file che contiene i messaggi tradotti, nell'ambito della configurazione locale, fa riferimento a un nome che viene definito «pacchetto», che di solito si sceglie opportunamente simile a quello del programma per il quale si fa la traduzione:

```
textdomain ("pippo");
```

L'esempio mostra l'istruzione da usare in un programma C per stabilire il nome del pacchetto secondo Gettext. Questo nome stabilisce che Gettext debba cercare il file '*sigla_locale*`/LC_MESSAGES/pippo.mo`'. Gettext determina il nome della prima parte del percorso, corrispondente a ciò che qui è stato mostrato con la metavariabile *sigla_locale*, analizzando alcune variabili di ambiente; precisamente segue questo ordine:

- '**LANGUAGE**'
- '**LC_ALL**'
- altre variabili '**LC_***'
- '**LANG**'

Dal valore contenuto in queste variabili si estrae la prima parte: quella che arriva fino al primo punto, se c'è. In pratica, se per ipotesi la variabile '**LANG**' contiene il valore '**it_IT.ISO-8859-1**', per Gettext è importante solo '**it_IT**'. Tuttavia, anche questa informazione tende a essere eccessiva, dal momento che contiene, oltre al linguaggio, anche l'area nazionale. In pratica, alla fine contano solo le prime due lettere, che esprimono il linguaggio in base allo standard ISO 639 (sezione 543).[2]

[2]Gettext analizza il contenuto delle variabili di ambiente perché con la funzione '**setlocale()**' è stata azzerata internamente la definizione '**LC_ALL**'. Usando la funzione '**setlocale()**' si potrebbe imporre un certo linguaggio, indipendentemente dalle variabili di ambiente relative.

Pertanto, tornando all'esempio iniziale, si tratta della directory '`it/LC_MESSAGES/pippo.mo`'. In condizioni normali, Gettext cerca questa directory a partire da '`/usr/share/locale/`' (o eventualmente un'altra posizione in base al modo in cui è stato compilato), tuttavia è possibile richiedere espressamente una collocazione differente attraverso un'istruzione già vista da collocare nel programma interessato:

```
bindtextdomain ("pippo", "/var/tmp");
```

In tal caso, se si scrive questo in un programma, Gettext andrà a cercare precisamente il file '`/var/tmp/it/LC_MESSAGES/pippo.mo`'.

315.4 Creazione e mantenimento dei file PO

È già stato mostrato in breve come si crea un file PO attraverso il programma '**`xgettext`**'. È il caso di osservare che '**`xgettext`**' può ricevere l'indicazione di più file sorgenti che fanno capo allo stesso dominio di traduzione:

`xgettext` [*opzioni*] *file_sorgente …*

In particolare, tra le opzioni può essere interessante segnalare '**`--default-domain=`***dominio*', che serve a '**`xgettext`**' per conoscere il dominio a cui si fa riferimento, creando così il file '*dominio* `.po`', invece del solito '`messages.po`'.

```
$ xgettext --default-domain=ciao *.c
```

L'esempio mostra come ottenere il file '`ciao.po`' a partire da tutti i file che terminano con l'estensione '`.c`'.

Quando si aggiornano i sorgenti di un programma già tradotto, si pone il problema di aggiornare nello stesso modo i file PO precedenti. Per fare questo si deve ricreare il file PO iniziale non tradotto, nel modo appena visto, quindi si usa il programma '**`msgmerge`**':

`msgmerge` [*opzioni*] *file_po_originale file_po_successivo* `>` *file_po_aggiornato*

In pratica, '**`msgmerge`**' fonde assieme due file PO, preservando le traduzioni del primo file riferite a messaggi che si trovano ancora nel secondo. Per esempio, se si dispone già del file '`vecchio.po`' con le traduzioni, mentre con '**`xgettext`**' è appena stato generato un file PO non tradotto per lo stesso programma, che qui viene chiamato '`non_tradotto.po`', si può ottenere un nuovo file PO con le traduzioni vecchie ancora valide e con i messaggi nuovi da tradurre:

```
$ msgmerge vecchio.po non_tradotto.po > nuovo.po
```

Naturalmente, questa operazione si fa nel momento in cui ci si accinge ad aggiornare materialmente la traduzione del programma, altrimenti questo lavoro non avrebbe senso, dal momento che un file PO contenente messaggi non tradotti non può essere compilato.

`msgfmt` [*opzioni*] *file_po*

Il programma '**`msgfmt`**' è quello che si occupa di compilare i file PO ottenendo i file MO, adatti alla propria piattaforma. È praticamente indispensabile utilizzare l'opzione '**`--output-file=`***file_mo*' ('**`-o`**'), per indicare il nome del file da creare. Inoltre, è opportuno utilizzare più di una volta l'opzione '**`--verbose`**' ('**`-v`**') per avere una visione chiara del procedimento, ovvero dei motivi per i quali alle volte il file non viene compilato.

```
$ msgfmt -vvvv --output-file=prova.mo prova.po
```

L'esempio mostra l'utilizzo tipico di questo programma, dove in particolare viene richiesto un livello di dettaglio delle informazioni generate molto elevato (quattro volte '**`-v`**').

315.4.1 Commenti «fuzzy»

Ogni volta che qualche indicazione all'interno di un file PO è incerta, in quanto predefinita o determinata automaticamente in modo non sicuro, viene aggiunto un commento speciale contenente la parola '**`fuzzy`**'. In presenza di commenti del genere si richiede un intervento manuale, dopo il quale deve essere rimossa tale parola, altrimenti '**`msgfmt`**' si rifiuta di completare la compilazione dei file PO.

315.5 Gettext con i programmi Perl

Esiste la possibilità di utilizzare Gettext anche nei programmi Perl. Per questo è necessario includere nel programma Perl il riferimento a un modulo esterno: Perl-gettext. Il tutto si svolge in maniera molto simile a un programma C, inserendo inizialmente le istruzioni seguenti:

```
use POSIX;
use Locale::gettext;
setlocale (LC_ALL, "");
textdomain ("dominio_gettext");
[bindtextdomain ("dominio_gettext", "directory");]
```

Per esempio, se si tratta del programma «Pippo», il dominio per Gettext potrebbe essere convenientemente «pippo», arrivando al risultato seguente:

```
use POSIX;
use Locale::gettext;
setlocale (LC_ALL, "");
textdomain ("pippo");
bindtextdomain ("pippo", "/opt/pippo/locale");
```

> Potrebbe essere che la funzione '**`bindtextdomain()`**' non si comporti come previsto; in tal caso sarebbe meglio evitarne l'uso.

Per il resto, tutto funziona come per i sorgenti scritti in C:

```
print STDOUT (gettext ("Hello world\n"));
```

Tuttavia, Perl non è identico al C, per cui occorre osservare alcune situazioni specifiche. In particolare, non è possibile inserire in un argomento della funzione '**`gettext()`**' una variabile di Perl che deve essere espansa, perché questa espansione avverrebbe prima che '**`gettext()`**' possa ricevere tale argomento. Pertanto, l'esempio seguente non può essere tradotto:

```
# Esempio errato.
print STDOUT (gettext ("Il file $file contiene caratteri non validi\n"));
```

Il modo giusto di agire è quello di sostituire '**`print()`**' con '**`printf()`**', come nell'esempio seguente:

```
# Esempio corretto.
printf STDOUT (gettext ("Il file %s contiene caratteri non validi\n"),
               $file);
```

Infatti, il parametro '**`%s`**' viene sostituito alla fine da '**`printf`**', per cui inizialmente la stringa non viene modificata.

Un altro problema da considerare sono i messaggi lunghi, che richiedono più righe. In Perl si potrebbe fare una cosa del genere:

```
    printf STDOUT
      (gettext
        ( "Usage: %s --input-type=TYPE INPUT_FILE REPORT_FILE\n"
```

```
        . "       %s --help\n"
        . "       %s --version\n"
        . "\n"
        . "Check for HTTP and FTP URI inside a text.\n"
        . "\n"
        . "Options:\n"
        . "--help              display this help and exit.\n"
        . "--version           display version information and exit.\n"
        . "--input-type=TYPE   define the input type:\n"
        . "    standard        input is a simple text file;\n"
        . "    html, sgml      input is a typical SGML file;\n"
        . "    texi, texinfo   input is a Texinfo source file.\n"
        . "\n"
        . "Arguments:\n"
        . "\n"
        . "INPUT_FILE          the input file.\n"
        . "\n"
        . "REPORT_FILE         a file that is generated with the reported\n"
        . "                      errors.\n"),
       $program_name, $program_name, $program_name);
```

Ma questo non viene riconosciuto da '**`xgettext`**' che riesce a prelevare solo la prima riga:

```
#: urichk:55
#, c-format
msgid "Usage: %s --input-type=TYPE INPUT_FILE REPORT_FILE\n"
msgstr ""
```

In queste situazioni eccezionali, occorre intervenire a mano nel file PO; sia la prima volta che si crea il file, sia tutte le volte successive in cui lo si aggiorna.

315.5.1 Alleviare gli inconvenienti di un modulo in più

Scrivere un programma Perl che faccia uso di Gettext, significa costringere a installare il modulo Perl-gettext. Purtroppo, una delle cose che complicano di più l'utilizzo di programmi Perl sono i moduli aggiuntivi necessari che devono essere installati perfettamente come previsto.

Questo potrebbe sembrare un problema secondario; invece non lo è affatto. A questo punto, se si vuole consentire al proprio programma Perl di funzionare anche in un ambiente non tanto amichevole, si deve prevedere una via di uscita:

```
#!/usr/bin/perl
#
#...

use POSIX;
use Locale::gettext;
setlocale (LC_ALL, "");
textdomain ("pippo");

#sub gettext
#{
#    return $_[0];
#}

#...
```

Come si vede nell'esempio, appare la dichiarazione di una funzione commentata, il cui scopo sarebbe quello di sostituirsi alla funzione '**`gettext()`**' del modulo '**`Locale::gettext`**'. Se non si dispone di Perl-gettext basta commentare la prima parte e togliere i commenti dalla seconda: ovviamente i messaggi rimarranno nella lingua di partenza.

```
#!/usr/bin/perl
#
```

```
#...

#use POSIX;
#use Locale::gettext;
#setlocale (LC_ALL, "");
#textdomain ("pippo");

sub gettext
{
    return $_[0];
}

#...
```

Indice analitico del volume

`/usr/lib/p2c/p2crc`, 3248
Basic, 3544, 3551
BC, 3522
Bubblesort, 3159, 3235, 3291, 3382, 3450, 3512, 3537
C, 3169
C: `calloc()`, 3209
C: `do-while`, 3183
C: `EOF`, 3212
C: `exit()`, 3190
C: `fclose()`, 3214
C: `fgets()`, 3218
C: `FILE`, 3212
C: `fopen()`, 3213
C: `for`, 3183
C: `fputs()`, 3219
C: `fread()`, 3216
C: `free()`, 3210
C: `fseek()`, 3217
C: `ftell()`, 3218
C: `fwrite()`, 3216
C: `if`, 3180
C: `malloc()`, 3209
C: `printf()`, 3188
C: `scanf()`, 3189
C: `switch`, 3181
C: `typedef`, 3207
C: `while`, 3182
C: `__DATE__`, 3224
C: `__FILE__`, 3223
C: `__LINE__`, 3223
C: `__STDC__`, 3224
C: `__TIME__`, 3224
C, preprocessore: `#define`, 3221, 3221
C, preprocessore: `#elif`, 3222
C, preprocessore: `#else`, 3222
C, preprocessore: `#endif`, 3222
C, preprocessore: `#ifdef`, 3222
C, preprocessore: `#ifndef`, 3222
C, preprocessore: `#if`, 3222
C, preprocessore: `#include`, 3220
C, preprocessore: `#line`, 3223
C, preprocessore: `#undef`, 3223
espressione regolare, 3335
file-make, 3242
GCC, 3193
GCC: ottimizzazione, 3193
GCJ, 3410
Gettext, 3556
Hanoi, 3160, 3237, 3293, 3383, 3452, 3514, 3538
Java, 3406, 3413

Java Development Kit, 3410
Java: `do-while`, 3422
Java: `for`, 3423
Java: `if`, 3420
Java: `super`, 3432
Java: `switch`, 3421
Java: `this`, 3433
Java: variabili, 3416
Java: `while`, 3422
JDK, 3410
Kaffe, 3406
Make, 3242
makefile, 3242
`p2c`, 3251
parametro attuale, 3266
parametro formale, 3264
Pascal, 3248, 3254, 3270, 3278
Pascal: array, 3270
Pascal: `case`, 3260
Pascal: `const`, 3273
Pascal: `for`, 3263
Pascal: funzione, 3263
Pascal: `if`, 3259
Pascal: insieme, 3274
Pascal: procedura, 3263
Pascal: `Read()`, 3267
Pascal: `Readln()`, 3267
Pascal: record, 3275
Pascal: `repeat-until`, 3262
Pascal: `set of`, 3274
Pascal: sottointervallo, 3274
Pascal: stringhe, 3271
Pascal: tipo enumerativo, 3273
Pascal: `type`, 3272
Pascal: `while`, 3262
Pascal: `Write()`, 3267
Pascal: `Writeln()`, 3267
Pascal-to-C, 3248
Perl, 3301
Perl: `abs()`, 3365
Perl: array, 3307
Perl: `atan2()`, 3365
Perl: `binmode`, 3354
Perl: `chdir()`, 3353
Perl: `chmod()`, 3349
Perl: `chomp()`, 3354
Perl: `chop()`, 3355
Perl: `chown`, 3350
Perl: `chr()`, 3366
Perl: `close`, 3355
Perl: `cos()`, 3365
Perl: `defined()`, 3367

Perl: delete(), 3368
Perl: die(), 3370
Perl: do(), 3370
Perl: eof, 3355
Perl: espressioni, 3313
Perl: espressioni regolari, 3335
Perl: eval(), 3370
Perl: exec(), 3362
Perl: exists(), 3368
Perl: exit(), 3371
Perl: exp(), 3365
Perl: fcntl, 3356
Perl: file, 3339
Perl: fileno, 3356
Perl: flock, 3356
Perl: for, 3318
Perl: foreach, 3319
Perl: getc, 3357
Perl: glob(), 3353
Perl: hash, 3311
Perl: hex(), 3366
Perl: if, 3316
Perl: int(), 3365
Perl: ioctl, 3357
Perl: keys(), 3369
Perl: kill(), 3362
Perl: link(), 3350
Perl: liste, 3307
Perl: log(), 3365
Perl: lstat(), 3350
Perl: m//, 3332
Perl: mkdir(), 3353
Perl: oct(), 3367
Perl: open(), 3357
Perl: operatori, 3313
Perl: ord(), 3367
Perl: pipe, 3359
Perl: pop(), 3369
Perl: print(), 3359
Perl: printf(), 3359
Perl: push(), 3369
Perl: q//, 3331
Perl: qq//, 3331
Perl: qw//, 3332
Perl: qx//, 3331
Perl: read(), 3359
Perl: readlink(), 3350
Perl: rename(), 3351
Perl: require(), 3371
Perl: rmdir(), 3354
Perl: s//, 3333
Perl: scalar(), 3367

Perl: scalare, 3303
Perl: `seek()`, 3360
Perl: `select()`, 3360
Perl: `sin()`, 3365
Perl: `sleep()`, 3363
Perl: `splice()`, 3369
Perl: `sprintf()`, 3360
Perl: `sqrt()`, 3366
Perl: `stat()`, 3351
Perl: stringhe, 3330
Perl: subroutine, 3322
Perl: `symlink()`, 3352
Perl: `system()`, 3363
Perl: `tell()`, 3361
Perl: `time()`, 3363
Perl: `times()`, 3364
Perl: `tr//`, 3334
Perl: `umask()`, 3364
Perl: `unless`, 3316
Perl: `unlink()`, 3352
Perl: `until`, 3317
Perl: `utime()`, 3352
Perl: variabili predefinite, 3303
Perl: `warn()`, 3371
Perl: `while`, 3317
Perl: `y//`, 3334
Perl: `-x`, 3348
programmazione: C, 3169
programmazione: Java, 3413
programmazione: Perl, 3301
programmazione: pseudocodifica, 3152
programmazione: Scheme, 3465, 3485, 3492
Quicksort, 3162, 3237, 3293, 3384, 3453, 3515, 3539
ricerca binaria, 3158
Scheme, 3458, 3465, 3485, 3492, 3499, 3502
`~/.kawarc.scm`, 3461
`~/.p2crc`, 3248
`$CLASSPATH`, 3435, 3460
`$P2CRC`, 3248
`$POSIXLY_CORRECT`, 3530

Appunti di informatica libera 2003.01.01

Volume VI

Linguaggi di programmazione specifici

Appunti Linux

Copyright © 1997-2000 Daniele Giacomini

Appunti di informatica libera

Copyright © 2000-2003 Daniele Giacomini

Via Morganella Est, 21 -- I-31050 Ponzano Veneto (TV) -- *daniele @ swlibero.org*

Le informazioni contenute in questa opera possono essere diffuse e riutilizzate in base alle condizioni poste dalla licenza GNU General Public License, come pubblicato dalla Free Software Foundation.

In caso di modifica dell'opera e/o di riutilizzo parziale della stessa, secondo i termini della licenza, le annotazioni riferite a queste modifiche e i riferimenti all'origine di questa opera, devono risultare evidenti e apportate secondo modalità appropriate alle caratteristiche dell'opera stessa. In nessun caso è consentita la modifica di quanto, in modo evidente, esprime il pensiero, l'opinione o i sentimenti del suo autore.

L'opera è priva di garanzie di qualunque tipo, come spiegato nella stessa licenza GNU General Public License.

Queste condizioni e questo copyright si applicano all'opera nel suo complesso, salvo ove indicato espressamente in modo diverso.

The informations contained inside this work can be spread and reused under the terms of the GNU General Public License as published by the Free Software Foundation.

If you modify this work and/or reuse it partially, under the terms of the license, the notices about these changes and the references about the original work, must be evidenced conforming to the work characteristics. IN NO EVENT IS ALLOWED TO MODIFY WHAT ARE CLEARLY THE THOUGHTS, THE OPINIONS AND/OR THE FEELINGS OF THE AUTHOR.

This work is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

These conditions and this copyright apply to the whole work, except where clearly stated in a different way.

Una copia della licenza GNU General Public License, versione 2, si trova nell'appendice A.

A copy of GNU General Public License, version 2, is available in appendix A.

The main distribution for *Appunti di informatica libera* is described below. For every distribution channel the maintainer's name and address is also reported.

Internet

- direct reading: <*http://a2.swlibero.org/*>
  download: <*ftp://a2.swlibero.org/a2/*> and <*http://a2.swlibero.org/ftp/*>
  Michele Dalla Silvestra, *mds @ swlibero.org*
- direct reading: <*http://appuntilinux.torino.linux.it/*>
  download: <*ftp://ftp.torino.linux.it/appunti-linux/*>
  Carlo Perassi, *carlo @ linux.it*
- direct reading: <*http://sansone.crema.unimi.it/linux/a2/HTML/*>
  download: <*http://sansone.crema.unimi.it/linux/a2/*>
  Fabrizio Zeno Cornelli, *zeno @ filibusta.crema.unimi.it*
- direct reading: <*http://www.pctime.it/servizi/appunti-linux/*>
  download: <*http://www.pctime.it/servizi/appunti-linux/a2-prelievo/*>
  Franco Lazzero, PCTIME, *pctime @ pctime.net*
- direct reading: <*http://www.a2.prosa.it/*>
  download: <*ftp://ftp.a2.prosa.it/*>
  Davide Barbieri, *paci @ prosa.it*
- direct reading: <*http://linux.pueste.it/*>
  download: <*http://linux.pueste.it/filearea/AppuntiLinux/*>
  David Pisa, *david @ iglu.cc.uniud.it*
- direct reading: <*http://www.informasiti.com/Appunti/HTML/*>
  download: <*http://www.informasiti.com/Appunti/*>
  Claudio Neri, Sincro Consulting, *neri.c @ sincroconsulting.com*

GNU distributions

- GNU/Linux **Debian** <*http://packages.debian.org/appunti-informatica-libera*>
  Massimo Dal Zotto, *dz @ cs.unitn.it*

Italian magazine's CD-ROM

- **inter-punto-net** <*http://www.interpuntonet.it*>
  Michele Dalla Silvestra, *mds @ swlibero.org*
- **Internet News** <*http://inews.tecnet.it*>
  Francesco Facconi, *francescofacconi @ libero.it*
  Fabio Ferrazzo, *fabio.fr @ tiscalinet.it*
- **Linux Magazine** <*http://www.edmaster.it/prodotti/linux/ult-riv.html*>
  Emmanuele Somma, *esomma @ ieee.org*

La diffusione di questa opera è incoraggiata in base ai termini della licenza.
The spread of this work is encouraged under the terms of the license.

Parte lxiii Linguaggi per la comparazione 3571
316 Espressioni regolari standard 3572
317 Confronto sintetico tra le espressioni regolari «reali» 3580
Parte lxiv Linguaggi per la scansione di file di testo 3583
318 SED: introduzione 3584
319 AWK: introduzione 3592
320 AWK: funzioni e array 3611
Parte lxv Linguaggi macro 3617
321 M4: introduzione 3618
Parte lxvi DBMS e SQL 3629
322 Introduzione ai DBMS 3631
323 Introduzione a SQL 3642
324 PostgreSQL: struttura e preparazione 3667
325 PostgreSQL: il linguaggio 3692
326 PostgreSQL: accesso attraverso PgAccess 3706
327 PostgreSQL: accesso attraverso WWW-SQL 3714
328 Le funzioni e i trigger in PostgreSQL: un’esercitazione didattica 3725
Indice analitico del volume 3732

Parte lxiii

Linguaggi per la comparazione

316 Espressioni regolari standard 3572

316.1 RE: BRE, ERE e SRE 3572
316.2 Problemi di localizzazione 3572
316.3 Composizione di un’espressione regolare e corrispondenza 3573
316.4 Espressioni tra parentesi quadre 3576
316.5 Precedenze 3579
316.6 Riferimenti 3579

317 Confronto sintetico tra le espressioni regolari «reali» 3580

Espressioni regolari standard

L'espressione regolare è un modo per definire la ricerca di stringhe attraverso un modello di comparazione. Viene usato da diversi programmi di servizio, ma non tutti aderiscono agli stessi standard. In questo capitolo si vuole descrivere lo standard POSIX al riguardo.

Questo tipo di definizione non vale in generale e non corrisponde nemmeno ad alcuna situazione pratica in cui vengono utilizzate le espressioni regolari con i programmi che si trovano generalmente con GNU/Linux. Tuttavia, è un riferimento utile per comprendere meglio la filosofia che sta alla base delle espressioni regolari.

Per studiare la grammatica delle espressioni regolari, occorre abbandonare qualunque resistenza, tenendo presente che l'interpretazione di queste espressioni va fatta da sinistra a destra; inoltre, ogni simbolo può avere un significato differente in base al contesto in cui si trova.

> Raramente si può affermare che un'espressione regolare sia «errata»; nella maggior parte dei casi in cui si commettono degli errori, si ottiene comunque qualcosa che può avere un significato (indipendentemente dal fatto che questa possa avere o meno una corrispondenza). È ancora più difficile che una realizzazione in cui si utilizzano le espressioni regolari sia in grado di segnalare un errore grammaticale nella loro scrittura.

316.1 RE: BRE, ERE e SRE

Un'espressione regolare, come definita nello standard POSIX 1003.2, può essere espressa attraverso due tipi di grammatiche differenti: le espressioni regolari di base, o elementari, identificate dall'acronimo BRE (*Basic regular expression*), e le espressioni regolari estese, identificate dall'acronimo ERE (*Extended regular expression*). La grammatica delle espressioni regolari tradizionali degli ambienti Unix viene identificata dall'acronimo SRE (*Simple regular expression*); in generale, per fare riferimento a espressioni regolari non meglio definite, si usa anche soltanto l'acronimo RE.

Attualmente si fa riferimento soltanto a espressioni regolari di tipo BRE o di tipo ERE, dipendendo dal programma di servizio la scelta tra l'una o l'altra forma. In generale, la necessità di definire un modello grammaticale differente da SRE dipende dalla presenza di problemi legati alla localizzazione.

316.2 Problemi di localizzazione

L'espressione regolare, essendo un mezzo per identificare una porzione di testo, risente di problemi legati alle definizioni locali degli insiemi di caratteri.

Per prima cosa occorre considerare che gli alfabeti nazionali sono differenti da un linguaggio all'altro. Dal punto di vista della localizzazione, gli elementi che compongono gli alfabeti sono degli *elementi di collazione* (*collating element*). Questi elementi compongono un insieme ordinato, definito *sequenza di collazione* (*collating sequence*), che permette di stabilire l'ordine alfabetico delle parole. In situazioni particolari, alcuni elementi di collazione sono rappresentati da più di un carattere, cosa che può dipendere da motivazioni differenti. Per fare un esempio comune, questo può essere causato dalla mancanza del carattere adatto a rappresentare un certo elemento, come succede nella lingua tedesca quando si utilizza un insieme di caratteri che non dispone delle vocali con la dieresi, oppure manca la possibilità di indicare la lettera «ß»:

```
ß       --> ss
ä       --> ae
ö       --> oe
ü       --> ue
```

Nella lingua tedesca, nel momento in cui si utilizzano le stringhe «ae», «oe», «ue» e «ss», in sostituzione delle lettere che invece avrebbero dovuto essere utilizzate, queste stringhe vanno considerate come la rappresentazione di tali lettere, costituendo così un elemento di collazione unico. Per esempio, in tedesco la parola «schal» viene prima di «schälen», anche se la seconda fosse scritta come «schaelen».

Ai fini della definizione di un'espressione regolare, questo fatto si traduce nella possibilità di fare riferimento a degli elementi di collazione attraverso la stringa corrispondente, nel momento in cui non è possibile, o non conviene usare il carattere che lo rappresenta simbolicamente in base a una codifica determinata. Tuttavia, il testo su cui si esegue la ricerca attraverso un'espressione regolare, viene interpretato a livello di carattere, per cui non è possibile identificare un elemento di collazione in una sottostringa composta da più caratteri. In pratica, un'espressione regolare non riuscirebbe a riconoscere la lettera «ä» nella parola '**schaelen**'.

Alcuni elementi di collazione possono essere classificati come equivalenti. Per esempio, nella lingua italiana le lettere «e», con o senza accento, rappresentano questo tipo di equivalenza. Gli elementi di collazione «equivalenti» costituiscono una *classe di equivalenza*.

Infine, i caratteri (e non più gli elementi di collazione) possono essere classificati in base a diversi altri tipi di sottoinsiemi, a cui si fa riferimento attraverso dei nomi standard. In generale si tratta di distinguere tra: lettere maiuscole, lettere minuscole, cifre numeriche, cifre alfanumeriche, ecc.

316.3 Composizione di un'espressione regolare e corrispondenza

Un'espressione regolare è una stringa di caratteri, che nel caso più semplice rappresentano esattamente la corrispondenza con la stessa stringa. All'interno di un'espressione regolare possono essere inseriti dei caratteri speciali, che permettono di rappresentare delle corrispondenze in situazioni più complesse. Per fare riferimento a tali caratteri in modo letterale, occorre utilizzare delle tecniche di protezione, che variano a seconda del contesto.

I caratteri speciali sono tali solo nel contesto per il quale sono stati previsti. Al di fuori di quel contesto possono essere caratteri normali, o caratteri speciali con un significato differente.

La corrispondenza tra un'espressione regolare e una stringa, quando avviene, serve a delimitare una sottostringa che può andare dalla dimensione nulla fino al massimo della stringa di partenza. È importante chiarire che anche la corrispondenza che delimita una stringa nulla può avere significato, in quanto identifica una posizione precisa nella stringa di partenza. In generale, se sono possibili delle corrispondenze differenti, viene presa in considerazione quella che inizia il più a sinistra possibile e si estende il più a destra possibile.

316.3.1 Ancoraggio iniziale e finale

In condizioni normali, un'espressione regolare può individuare una sottostringa collocata in qualunque posizione della stringa di partenza. Per indicare espressamente che la corrispondenza deve iniziare obbligatoriamente dall'inizio della stringa, oppure che deve finire esattamente alla fine della stringa stessa, si usano due ancore, rappresentate dai caratteri speciali '**^**' e '**$**', ovvero dall'accento circonflesso e dal dollaro.

Per la precisione, un accento circonflesso che si trovi **all'inizio** di un'espressione regolare identifica la sottostringa nulla che si trova idealmente all'inizio della stringa da analizzare; nello stesso modo, un dollaro che si trovi **alla fine** di un'espressione regolare identifica la sottostringa nulla che si trova idealmente alla fine della stringa stessa. Nel caso particolare delle espressioni regolari BRE, i caratteri '**^**' e '**$**' hanno questo significato anche nell'ambito di una sottoespressione, all'inizio o alla fine della stessa. Una sottoespressione è una porzione di espressione regolare delimitata nel modo che verrà mostrato in seguito.

Per fare un esempio, l'espressione regolare '**^ini**' corrisponde alla sottostringa '**ini**' della stringa '**inizio**'. Nello stesso modo, l'espressione regolare '**ini$**' corrisponde alla sottostringa '**ini**' della stringa '**scalini**'.

Un'espressione regolare può contenere entrambe le ancore di inizio e fine stringa. In tal caso si cerca la corrispondenza con tutta la stringa di partenza.

316.3.2 Delimitazione di una o più sottoespressioni

Una sottoespressione è una porzione di espressione regolare individuata attraverso dei delimitatori opportuni. Per la precisione, si tratta di parentesi tonde normali nel caso di espressioni regolari ERE (estese), oppure dei simboli '**\(**' e '**\)**' nel caso di espressioni regolari BRE.

La delimitazione di sottoespressioni può servire per regolare la precedenza nell'interpretazione delle varie parti dell'espressione regolare, oppure per altri scopi che dipendono dal programma in cui vengono utilizzate. In generale, dovrebbe essere ammissibile la definizione di sottoespressioni annidate.

Per fare un esempio, l'espressione regolare BRE '**\(anto\)logia**' corrisponde a una qualunque sottostringa '**antologia**'. Nello stesso modo funziona l'espressione regolare ERE '**(anto)logia**'.

316.3.3 Riferimento a una sottoespressione precedente (solo BRE)

Nelle espressioni regolari di tipo BRE è possibile utilizzare la forma '*n*', dove *n* è una cifra numerica da uno a nove, per indicare la corrispondenza con l'*n*-esima sottoespressione precedente. Per esempio, l'espressione regolare '**\(sia\) questo \1**' corrisponde alla sottostringa '**sia questo sia**' di un testo che può essere anche più lungo.

È importante osservare che la corrispondenza della forma '*n*' rappresenta ciò che è stato trovato effettivamente attraverso la sottoespressione, mentre se si volesse semplicemente ripetere lo stesso modello, basterebbe riscriverlo tale e quale.

316.3.4 Sottoespressioni alternative (solo ERE)

Esclusivamente nelle espressioni regolari ERE (estese), è possibile indicare la corrispondenza alternativa tra due modelli utilizzando il carattere speciale '**|**' (la barra verticale). Di solito si utilizza questa possibilità delimitando espressamente le sottoespressioni alternative, in modo da evitare ambiguità, tuttavia questo non dovrebbe essere necessario, dal momento che si tratta di un operatore con un livello molto basso di precedenza.

Per esempio, l'espressione regolare '**((auto)|(dog))matico**' può corrispondere indifferentemente alla sottostringa '**automatico**' oppure '**dogmatico**'.

316.3.5 Corrispondenza con un carattere singolo

In un'espressione regolare, qualsiasi carattere che nel contesto non abbia un significato particolare, corrisponde esattamente a se stesso.

Il carattere speciale '**.**' (il punto), rappresenta un carattere qualunque, a esclusione di *<NUL>*. Per esempio, l'espressione regolare '**nuo.o**' corrisponde a '**nuoto**', '**nuovo**' e ad altre sottostringhe simili. Per indicare un punto letterale, occorre utilizzare l'espressione '**\.**' (barra obliqua inversa, punto).

È possibile definire anche la corrispondenza con un carattere scelto tra un insieme preciso, utilizzando una notazione speciale, ovvero un'*espressione tra parentesi quadre* :

[*elenco_corrispondente*]
[^*elenco_non_corrispondente*]

Come si vede dallo schema sintattico, si distinguono due situazioni fondamentali: nel primo caso si definisce un elenco di corrispondenze; nel secondo si ottiene questa definizione indicando un elenco di caratteri che non si vogliono trovare. Si osservi che per negare l'elenco di corrispondenze si utilizza l'accento circonflesso, che quindi assume qui un significato speciale, differente dall'ancora di inizio già descritta.

L'elenco tra parentesi quadre può essere un elenco puro e semplice di caratteri (lungo a piacere), per cui, per esempio, l'espressione regolare '**piccol[aieo]**' corrisponde indifferentemente alle sottostringhe '**piccola**', '**piccoli**', '**piccole**' e '**piccolo**'. In alternativa può essere rappresentato attraverso uno o più intervalli di caratteri, ma questo implica delle complicazioni che verranno descritte in seguito.

Per negare un elenco, lo si fa precedere da un accento circonflesso. Per esempio, l'espressione regolare '**aiut[^ia]**' può corrispondere alla sottostringa '**aiuto**' e anche a molte altre, ma non può corrispondere né ad '**aiuti**', né ad '**aiuta**'.

Dal momento che l'accento circonflesso ha un significato speciale se appare all'inizio di tale contesto, questo può essere usato in modo letterale solo in una posizione più avanzata.

Infine, per indicare una parentesi quadra aperta letterale in un contesto normale, al di fuori delle espressioni tra parentesi quadre, basta l'espressione '**\[**'.

316.3.6 Corrispondenze multiple

Alcuni caratteri speciali fungono da operatori che permettono di definire e controllare il ripetersi di un modello riferito a un carattere precedente, a una sottoespressione precedente, oppure a un riferimento all'indietro delle espressioni regolari BRE.

In tutti i tipi di espressione regolare, l'asterisco ('*****') corrisponde a nessuna o più ripetizioni di ciò che gli precede. Per esempio, l'espressione regolare '**aiuto***' corrisponde alla sottostringa

‘**aiut**’, oppure ‘**aiuto**’, come anche ad ‘**aiutooooooo**’, ecc. Inoltre, è il caso di osservare che l’espressione regolare ‘**.***’ corrisponde a qualunque stringa, di qualunque dimensione.

Per indicare un asterisco letterale in un contesto normale, basta farlo precedere da una barra obliqua inversa: ‘*****’.

Nel caso di espressioni regolari ERE si possono utilizzare anche gli operatori ‘**+**’ e ‘**?**’, per indicare rispettivamente una o più occorrenze dell’elemento precedente, oppure zero o al massimo un’occorrenza di tale elemento. Per esempio, l’espressione regolare ‘**aiuto+**’ corrisponde alla sottostringa ‘**aiuto**’, oppure ‘**aiutoo**’, ‘**aiutooooo**’, ecc., mentre l’espressione regolare ‘**aiuto?**’ può corrispondere alla sottostringa ‘**aiut**’, oppure ‘**aiuto**’.

Le espressioni regolari BRE e ERE permettono l’utilizzo di un’altra forma più precisa e generalizzata per esprimere la ripetizione di qualcosa. Nel caso di BRE si usano i modelli

\{*n*\}
\{*n*,\}
\{*n*,*m*\}

mentre nel caso di ERE si usano forme equivalenti senza le barre oblique inverse:

{*n*}
{*n*,}
{*n*,*m*}

Si tenga presente che *n* rappresenta un numero non negativo, mentre *m*, se utilizzato, deve essere un numero maggiore di *n*.

Nella prima delle tre forme, si intende indicare la ripetizione di *n* volte esatte l’elemento precedente; nella seconda si intendono almeno *n* volte; nella terza si intendono tante ripetizioni da *n* a *m*. In generale, per garantire che un’espressione regolare sia portabile, occorre che il limite massimo rappresentato da *m* non superi 255.

316.4 Espressioni tra parentesi quadre

Si è accennato all’uso delle espressioni tra parentesi quadre, per indicare la scelta tra un elenco di caratteri, o tra tutti i caratteri esclusi quelli dell’elenco. Un’espressione del genere si traduce sempre nella corrispondenza con un carattere singolo. All’interno di un’espressione del genere, si possono utilizzare forme particolari per indicare un carattere, attraverso un simbolo di collazione, una classe di equivalenza, oppure attraverso una classe di caratteri. È molto importante anche la possibilità di definire degli intervalli, che è stata saltata volutamente nella descrizione precedente di queste espressioni.

316.4.1 Corrispondenza con un elemento di collazione

Se si hanno difficoltà a indicare dei caratteri in un’espressione tra parentesi quadre, potrebbe essere opportuno indicarli attraverso l’elemento di collazione corrispondente. Supponendo che nella localizzazione utilizzata esista l’elemento di collazione ‘**ä**’, identificato dal simbolo di collazione ‘**<a:>**’, mancando la possibilità di usare il carattere corrispondente, questo si potrebbe esprimere nella forma ‘**[.a:.]**’.

In generale, è possibile indicare un carattere singolo all’interno dei delimitatori ‘**[.**’ e ‘**.]**’, come se fosse un elemento di collazione. Per esempio, ‘**[.a.]**’ è perfettamente uguale all’espressione ‘**a**’. In questo modo, si può usare la tecnica di rappresentazione degli elementi di collazione quando il contesto rende difficile l’indicazione di qualche carattere.

È necessario ribadire che il simbolo di collazione può apparire solo all'interno di un'espressione tra parentesi quadre. Per fare un esempio pratico, trovandoci in una localizzazione adatta, volendo scrivere un'espressione regolare che corrisponda alla sottostringa '**schälen**', non potendo rappresentare il carattere '**ä**' si dovrebbe scrivere: '**sch[[.a:.]]len**', dove '**[.a:.]**' si sostituisce al carattere '**ä**', avendo definito che il simbolo di collazione per questo è '**<a:>**'.

316.4.2 Corrispondenza con una classe di equivalenza

Nell'ambito della sequenza di collazione della localizzazione che si usa, alcuni elementi possono essere considerati equivalenti ai fini dell'ordinamento. Questi elementi costituiscono una classe di equivalenza. All'interno di un'espressione tra parentesi quadre, per fare riferimento a un elemento qualunque di una certa classe di equivalenza, basta indicare uno di questi tra i delimitatori '**[=**' e '**=]**'. Per esempio, se si suppone che le lettere '**e**', '**è**' ed '**é**', appartengono alla stessa classe di equivalenza, per indicare indifferentemente una di queste, basta la notazione '**[=e=]**'.

Per indicare effettivamente una classe di equivalenza in un'espressione regolare, occorre ricordare che questa va inserita all'interno di un'espressione tra parentesi quadre. In pratica, l'espressione regolare che corrisponde indifferentemente alla stringa '**e**', '**è**' o '**é**', è '**[[=e=]]**'. Si osservi che in alternativa si poteva scrivere anche '**[eèé]**'.

316.4.3 Corrispondenza con una classe di caratteri

Nell'ambito della localizzazione, sono definiti alcuni gruppi di caratteri, attraverso l'uso di parole chiave standard. Per esempio: '**alpha**' definisce l'insieme delle lettere alfabetiche; '**digit**' definisce l'insieme delle cifre numeriche; '**space**' definisce l'insieme dei caratteri che visivamente si traducono in uno spazio di qualche tipo. Oltre a queste, sono definiti dei raggruppamenti, come nel caso di '**alnum**' che indica l'insieme di '**alpha**' e '**digit**'.

All'interno di un'espressione tra parentesi quadre, per indicare una classe di caratteri, si usa il nome riconosciuto dalla localizzazione, racchiuso tra i delimitatori '**[:**' e '**:]**'. Per esempio, per ottenere la corrispondenza con una sottostringa del tipo '**file***n*', dove *n* può essere una cifra numerica qualunque, si può utilizzare l'espressione regolare '**file[[:digit:]]**'.

La tabella 316.1 riepiloga i nomi delle classi di caratteri riconosciuti normalmente dalle localizzazioni (si veda anche la pagina di manuale *locale(5)*).

Tabella 316.1. Elenco dei nomi standard attribuiti alle classi di caratteri.

Classe di caratteri	Descrizione
upper	Collezione alfabetica delle lettere maiuscole.
lower	Collezione alfabetica delle lettere minuscole.
alpha	Lettere alfabetiche: di solito l'unione di '**upper**' e '**lower**'.
digit	Cifre numeriche.
alnum	Cifre alfanumeriche: di solito l'unione di '**alpha**' e '**digit**'.
punct	I caratteri di punteggiatura.
space	I caratteri definiti come «spazi bianchi» per qualche motivo.
blank	Di solito comprende solo '**<space>**' e '**<tab>**'.
cntrl	I caratteri di controllo che non possono essere rappresentati.
graph	Caratteri grafici: di solito l'unione di '**alnum**' e '**punct**'.
print	Caratteri stampabili: di solito l'insieme di '**alnum**', '**punct**' e di '**<space>**'.
xdigit	Cifre numeriche e alfabetiche per rappresentare numeri esadecimali.

316.4.4 Intervalli di caratteri

All'interno di un'espressione tra parentesi quadre, possono apparire anche degli intervalli di caratteri, includendo eventualmente anche gli elementi di collazione. Al contrario, non si possono usare le classi di equivalenza e nemmeno le classi di caratteri per indicare degli intervalli, perché non si traducono in un carattere preciso nell'ambito della codifica. La forma per esprimere un intervallo è la seguente:

inizio **-** *fine*

Questo lascia intendere che il trattino ('**-**') abbia un significato particolare all'interno di un'espressione tra parentesi quadre. Per fare un esempio molto semplice, l'espressione regolare '**[a-d]**' rappresenta un carattere compreso tra '**a**' e '**d**', in base alla localizzazione.

Gli intervalli si possono mescolare con gli elenchi e anche con altri intervalli. Per esempio, l'espressione regolare '**[a-dhi]**' individua un carattere compreso tra '**a**' e '**d**', oppure anche '**h**' o '**i**'.

Possono essere aggregati più elenchi assieme, ma tutti questi devono avere un inizio e una fine indipendente. Per esempio, l'espressione regolare '**[a-cg-z]**' rappresenta due intervalli, rispettivamente tra '**a**' e '**c**', e tra '**g**' e '**z**'. Al contrario, l'espressione regolare '**[a-c-z]**' indica l'intervallo da '**a**' a '**c**', oppure il trattino (perché è fuori dal contesto previsto per indicare un intervallo), oppure '**z**'.

> Quando si indicano degli intervalli non tanto «ovvi», occorre prestare attenzione alla localizzazione per sapere esattamente cosa viene coinvolto. In generale, per questo motivo, le espressioni regolari che contengono espressioni tra parentesi quadre con l'indicazioni di intervalli, non sono portabili da un sistema all'altro.

316.4.5 Protezione all'interno di espressioni tra parentesi quadre

Dal momento che in un'espressione tra parentesi quadre i caratteri '**^**', '**-**' e '**]**', hanno un significato speciale, per poterli utilizzare, occorrono degli accorgimenti: se si vuole usare l'accento circonflesso in modo letterale, è necessario che questo non sia il primo; per indicare il trattino si può descrivere un intervallo, in cui sia posto come carattere iniziale o finale. In alternativa, i caratteri che non si riescono a indicare (come le parentesi quadre), possono essere racchiuse attraverso i delimitatori dei simboli di collazione: '**[.[.]**' e '**[.].]**' per le parentesi e '**[.-.]**' per un trattino.

> All'interno di un'espressione tra parentesi quadre, i caratteri che sono speciali al di fuori di questo contesto, qui perdono il loro significato particolare (come nel caso del punto e dell'asterisco ('*****'), oppure ne acquistano uno nuovo (come nel caso dell'accento circonflesso).

316.5 Precedenze

Dopo la difficoltà che si affronta per comprendere il funzionamento delle espressioni regolari, l'ordine in cui le varie parti di queste vengono risolte, dovrebbe essere abbastanza intuitivo. La tabella 316.2 riassume questa sequenza, distinguendo tra espressioni BRE e ERE.

Tabella 316.2. Ordine di precedenza, dal più alto al più basso.

Tipo di componente l' espressione	Operatore BRE	Operatore ERE
Contenuto delle espressioni tra parentesi quadre.	[==] [::] [..]	[==] [::] [..]
Caratteri speciali resi letterali.	*carattere_speciale*	*carattere_speciale*
Espressioni tra parentesi quadre.	[]	[]
Sottoespressioni e riferimenti all'indietro (BRE).	\(\) *n*	
Raggruppamenti (ERE).		()
Ripetizioni.	* \{*m*,*n*\}	* + ? {*m*,*n*}
Concatenamento di espressioni (non si usano simboli).		
Ancore iniziali e finali.	^ $	^ $
Alternanza (solo ERE).		|

316.6 Riferimenti

- *regexp(M)*

  <http://www.lctn.com/~rjones/man/regexp.M.html>

- The Open Group, *The Single UNIX ® Specification, Version 2, Regular Expressions*, 1997

  <http://www.opengroup.org/onlinepubs/7908799/xbd/re.html>

Confronto sintetico tra le espressioni regolari «reali»

Date le diversità notevoli tra tutti i tipi di espressione regolare che si utilizzano in pratica con i programmi che ne fanno uso, vale la pena di riepilogare le differenze fondamentali tra lo standard POSIX e le realtà più importanti. In questo capitolo si raccolgono solo alcune tabelle di comparazione, che mostrano l'abbinamento tra diversi modelli di espressione compatibili. Le descrizioni sono scarne, tuttavia quello che si vede dovrebbe servire per collegare le cose, permettendo di comprendere quali sono le estensioni di ogni realizzazione.

Tabella 317.1. Confronto tra gli operatori fondamentali.

	BRE POSIX
escape	\
ancora	^
ancora	$
alternativa	
raggruppamento	\(\)
elenco	[ ]
riferimento	*n*

Tabella 317.2. Confronto tra gli operatori interni alle espressioni tra parentesi quadre.

	BRE POSIX
sequenze	*xy*...
intervalli	*x*-*y*
elementi di collazione	[. .]
caratteri equivalenti	[= =]
classi di caratteri	[: :]

Tabella 317.3. Simboli speciali.

	BRE POSIX
	[[:alnum:]_]
	[^[:alnum:]_]
inizio di parola	
fine di parola	
inizio o fine parola	
interno di una parola	
	[[:blank:]]
	[^[:blank:]]
	[[:digit:]]
	[^[:digit:]]

Tabella 317.4. Operatori di ripetizione.

	BRE POSIX
	$x*$
il minimo di $x*$	
il minimo di $x?$	
il minimo di $x+$	
	$x\{n\}$
	$x\{n,\}$
il minimo di $x\{n,\}$	
	$x\{n,m\}$
il minimo di $x\{n,m\}$	

In generale, si può osservare che i programmi GNU e Perl non permettono l'indicazione di simboli di collazione e nemmeno di classi di equivalenza. Inoltre, Perl non dispone nemmeno delle classi di caratteri. Per ovviare a questi inconvenienti, si utilizzano invece delle sequenze di escape.

A differenza di ciò che si vede di solito, Perl introduce un concetto nuovo: la corrispondenza minima di un'espressione regolare. Questo può essere molto importante in Perl, quando si delimitano delle sottoespressioni per estrapolare delle parti differenti di una stringa.

Parte lxiv

Linguaggi per la scansione di file di testo

318 SED: introduzione 3584

318.1 Avvio dell'eseguibile 3584
318.2 Logica di funzionamento 3585
318.3 Script e direttive multiple 3586
318.4 Direttive 3587
318.5 Esempi 3590
318.6 Riferimenti 3591

319 AWK: introduzione 3592

319.1 Principio di funzionamento e struttura fondamentale 3592
319.2 Avvio dell'interprete 3596
319.3 Espressioni 3597
319.4 Istruzioni 3603
319.5 Variabili predefinite 3608
319.6 Esempi 3609
319.7 Riferimenti 3610

320 AWK: funzioni e array 3611

320.1 Dichiarazione di funzioni 3611
320.2 Array 3612

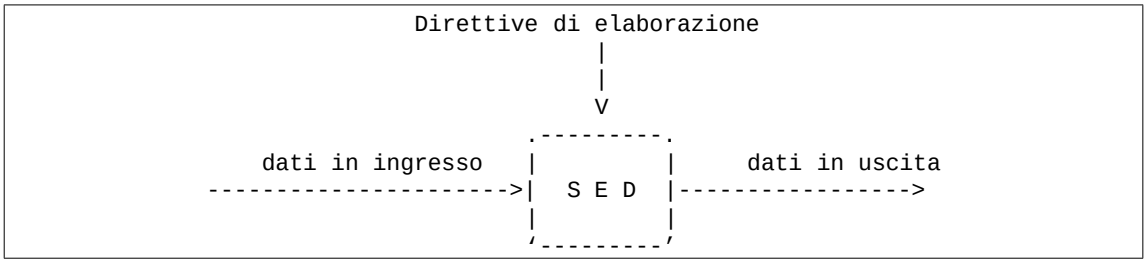
SED: introduzione

SED è un programma in grado di eseguire delle trasformazioni elementari in un flusso di dati di ingresso, proveniente indifferentemente da un file o da una pipeline. Questo flusso di dati viene letto sequenzialmente e la sua trasformazione viene restituita attraverso lo standard output.

Il nome è l'abbreviazione di Stream Editor, che descrive istantaneamente il senso di questo programma: *editor* di flusso. Volendo usare altri termini, lo si potrebbe definire come un programma per la modifica sequenziale di un flusso di dati espressi in forma testuale.

Volendo vedere SED come una scatola nera, lo si può immaginare come un oggetto che ha due ingressi: un flusso di dati in ingresso, composto da uno o più file di testo concatenati assieme; un flusso di istruzioni in ingresso, che compone il programma dell'elaborazione da apportare ai dati; un flusso di dati in uscita che rappresenta il risultato dell'elaborazione.

Figura 318.1. Flussi di dati che interessano SED.



In linea di principio, SED non consente di indicare dei caratteri speciali nei suoi comandi attraverso delle sequenze di escape.

318.1 Avvio dell'eseguibile

SED è costituito in pratica dall'eseguibile '**sed**', il quale interpreta un programma scritto in un linguaggio apposito, che gli viene fornito come argomento della riga di comando, o in un file.

`sed` [*opzioni*] [*programma_di_elaborazione*] [*file…*]

Il testo del programma, o il nome del file che lo contiene, può essere indicato attraverso delle opzioni adatte, oppure, in loro mancanza, può essere indicato come primo degli argomenti che seguono le opzioni. Alla fine possono essere indicati i file da elaborare e in loro mancanza si usa lo standard input.

Alcune opzioni

`-e` *istruzioni*

`--expression=`*istruzioni*

Questa opzione, che può essere utilizzata anche più volte, permette di specificare delle istruzioni SED che si aggiungono alle altre eventualmente già indicate.

`-f` *file_delle_istruzioni*

`--file` *file_delle_istruzioni*

Questa opzione permette di indicare un file contenente una serie di istruzioni SED. Anche questa opzione può essere usata più volte, aggiungendo ogni volta altre istruzioni al programma globale.

`-n` | `--quiet` | `--silent`

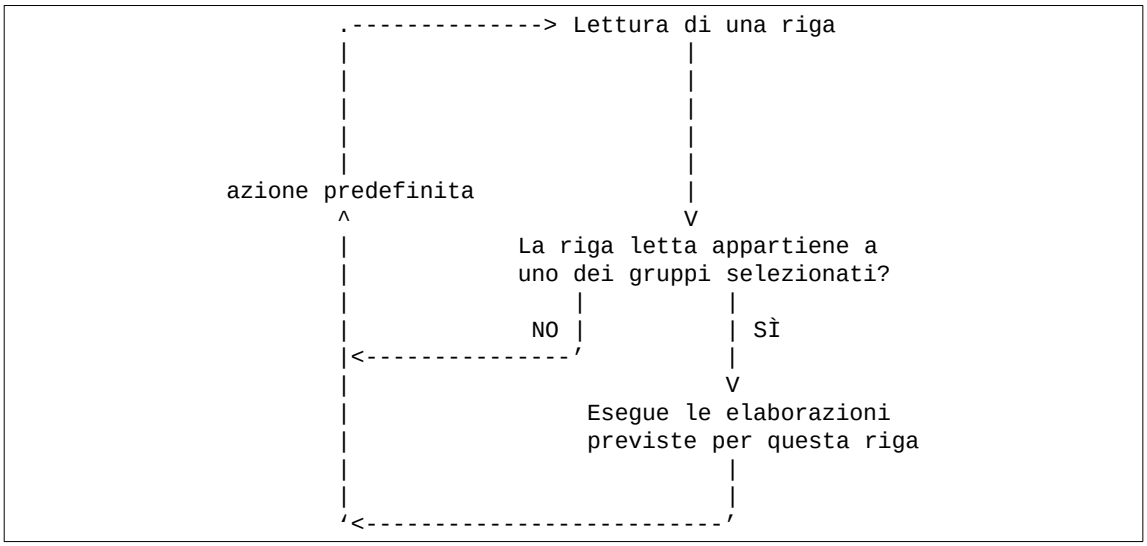
In condizioni normali, alla fine di ogni ciclo, SED emette il contenuto di quello che viene definito come *pattern space*. In pratica, ogni riga letta ed elaborata viene emessa attraverso lo standard output senza bisogno di un comando apposito. Utilizzando questa opzione, si fa in modo di evitare tale comportamento, così che il programma di elaborazione interpretato da SED deve ordinare quando emettere ogni riga.

318.2 Logica di funzionamento

Il primo compito di SED, una volta avviato, è quello di raccogliere tutto ciò che deve andare a comporre il programma di elaborazione: può trattarsi di direttive fornite singolarmente attraverso l'opzione '**-e**' e di gruppi di direttive fornite all'interno di file appositi, indicati attraverso l'opzione '**-f**'. In particolare, SED si prende cura di mantenerne intatto l'ordine. Successivamente, concatena i dati in ingresso secondo la sequenza indicata dei file posti alla fine della riga di comando, oppure utilizza direttamente lo standard input.

Lo schema che appare nella figura 318.2 si avvicina all'idea del funzionamento di SED: il flusso in ingresso viene letto sequenzialmente, una riga alla volta; ogni volta la riga viene messa in un'area transitoria, nota come *pattern space*; viene confrontata la riga con ogni direttiva del programma di elaborazione e se nessuna di queste direttive coincide, la riga non viene elaborata, compiendo semplicemente l'azione predefinita prima di passare al prossimo ciclo di lettura. Se una o più direttive del programma di elaborazione corrispondono alla riga, vengono eseguite sequenzialmente le elaborazioni previste; poi, alla fine, si passa comunque per l'esecuzione dell'azione predefinita.

Figura 318.2. Struttura semplificata del funzionamento di SED.



L'azione predefinita di SED è l'emissione del contenuto dell'area transitoria, per cui, se non venisse fornita alcuna direttiva a SED, si otterrebbe almeno la riemissione completa dello stesso file ricevuto in ingresso:

```
$ sed "" pippo.txt
```

L'esempio mostra proprio l'avvio dell'eseguibile '**sed**' allo scopo di interpretare una direttiva nulla, fornendo il file '`pippo.txt`' in ingresso. Il risultato è la riemissione del contenuto di questo file attraverso lo standard output.

Per impedire che questa azione si compia automaticamente, si utilizza l'opzione '**-n**' (ovvero '**--quiet**' o '**--silent**'). In questo modo, è compito delle direttive del programma di elaborazione il richiedere espressamente l'emissione della riga elaborata.

SED dispone di due aree transitorie per le elaborazioni: una che contiene la riga letta, che è già stata indicata; l'altra, definita come *hold space*, viene gestita eventualmente attraverso le direttive del programma di elaborazione interpretato da SED. L'utilizzo di questa seconda area di memoria non viene mostrato in questo capitolo.

Dal momento che SED è un programma storico dei sistemi Unix, è bene tenere presente le limitazioni che potrebbe avere in questo o quel sistema. In particolare, qualche realizzazione di SED potrebbe porre un limite alla dimensione delle righe. Questo fatto va tenuto presente quando si vogliono realizzare dei programmi «portabili», ovvero, da usare su piattaforme diverse, con sistemi operativi diversi.

Tabella 318.1. Riepilogo delle espressioni regolari di SED (BRE).

POSIX	GNU	Descrizione
.	.	Un carattere qualsiasi.
\	\	Escape.
^	^	Inizio riga.
$	$	Fine riga.
	|	Alternativa.
	\(\)	Raggruppamento.
\n	\n	Riferimento.
x*	x*	Zero o più caratteri qualsiasi.
	x\?	Zero o al massimo un carattere qualsiasi.
	x\+	Uno o più caratteri qualsiasi
x\{n\}	x\{n\}	Esattamente *n* volte *x*.
x\{n,\}	x\{n,\}	Almeno *n* volte *x*.
x\{n,m\}	x\{n,m\}	Da *n* a *m* volte *x*.
x\{,m\}	x\{,m\}	Da zero a *m* volte *x*.
[ ]	[ ]	Elenco.
xy...	xy...	Sequenze.
x-y	x-y	Intervalli.
[. .]		Elementi di collazione.
[= =]		Caratteri equivalenti.
[: :]	[: :]	Classi di caratteri.

318.3 Script e direttive multiple

Di solito, si vede utilizzare SED con direttive fornite direttamente attraverso la stessa riga di comando. Volendo realizzare un programmino un po' più complesso, si potrebbe scrivere direttamente uno script che deve essere interpretato direttamente da SED. Per farlo, occorre iniziare il file in questione con una delle due intestazioni seguenti:

```
#!/bin/sed -f
```

```
#!/bin/sed -nf
```

Nel primo caso, si fa in modo di fornire all'eseguibile '**`sed`**' (si suppone che si trovi nella directory '`/bin/`') l'opzione '**`-f`**', in modo che il file stesso venga inteso correttamente come un programma di elaborazione; nel secondo, oltre a questo, viene aggiunta l'opzione '**`-n`**', con la quale si inibisce l'emissione predefinita delle righe dopo ogni ciclo di elaborazione.[1]

Per quanto riguarda le direttive contenute nei file, queste utilizzano una riga per ognuna, dove le righe bianche o vuote vengono ignorate, assieme ai commenti che iniziano con il simbolo '`#`':

[1] È bene osservare che in uno script del genere non è possibile fare riferimento alle variabili di ambiente.

```
direttiva_di_elaborazione
direttiva_di_elaborazione
...
```

Le direttive fornite attraverso la riga di comando sono solitamente istruzioni singole; per cui, volendo aggiungerne delle altre, si utilizzano più opzioni '**-e**':

```
sed -e direttiva_di_elaborazione [-e direttiva_di_elaborazione]... file_in_ingresso...
```

Tuttavia, di solito è possibile indicare più direttive con una sola opzione '**-e**', separandole con un punto e virgola:

```
sed -e direttiva_di_elaborazione[;direttiva_di_elaborazione]... file_in_ingresso...
```

L'uso di più direttive nella riga di comando, con o senza il punto e virgola, è sconsigliabile in generale, dal momento che dovendo scrivere un programma di elaborazione complesso è preferibile usare un file, trasformandolo eventualmente in uno script come è stato mostrato all'inizio di questa sezione.

318.4 Direttive

Ogni direttiva di un programma di elaborazione SED fa riferimento, esplicitamente o implicitamente, a un gruppo di righe, identificate in qualche modo, a cui vengono applicati dei comandi.

```
[selezione_righe]comando
```

Il modello sintattico mostra l'indicazione di un comando dopo la selezione delle righe; questo comando può essere un raggruppamento di comandi, indicato all'interno di parentesi graffe.

318.4.1 Selezione delle righe

La selezione delle righe per una direttiva SED è il primo elemento importante per queste. La mancanza dell'indicazione di questa selezione rappresenta implicitamente la selezione di tutte le righe.

È importante osservare che le righe possono essere indicate anche attraverso la corrispondenza con un'espressione regolare, che comunque non deve essere confusa con i comandi che a loro volta possono avere a che fare con altre espressioni regolari.

Inoltre, è necessario ricordare che SED numera le righe a partire dalla prima del primo file, continuando fino alla fine dell'ultimo file, senza interrompere la numerazione.

- ```
  n
  ```

  Un numero puro e semplice, indica precisamente la riga *n*-esima.

- ```
  $
  ```

  Un dollaro rappresenta l'ultima riga dell'ultimo file.

- ```
  /espressione_regolare_elementare/
  ```

  Un'espressione regolare elementare (BRE), racchiusa tra due barre oblique normali, serve a selezionare tutte le righe per cui corrisponde questo modello. Dal momento che la barra obliqua viene usata come delimitatore, se questa deve essere inserita nel modello, occorre proteggerla con una barra obliqua inversa ('\/').
  ```

- *\xespressione_regolare_elementarex*

  Si tratta sempre della selezione delle righe in base alla corrispondenza con un'espressione regolare, con la differenza che questa viene delimitata con un carattere differente, *x*, scelto liberamente, in modo da non interferire con i simboli usati nel modello. Se il modello dell'espressione regolare dovesse contenere anche questo carattere usato per la delimitazione, potrebbe essere protetto con l'aggiunta della barra obliqua inversa all'inizio ('*x*').

- *riga_iniziale* , *riga_finale*

  È possibile indicare un intervallo di righe, unendo assieme due riferimenti a righe, sia in forma numerica che attraverso le espressioni regolari. Per quanto riguarda l'individuazione della prima riga dell'intervallo, la cosa è abbastanza semplice; in particolare, se si tratta di un'espressione regolare, la prima corrispondenza indica la prima riga. Più complicato è il modo in cui viene preso in considerazione il secondo modello:

  – se si tratta di un numero, questo rappresenta l'*n*-esima riga da raggiungere, che deve essere considerata inclusa nell'intervallo, ma se questo numero indica una riga precedente alla riga iniziale dell'intervallo, allora viene selezionata solo quella iniziale;
  – se si tratta di un'espressione regolare, allora questo modello viene confrontato a partire dalla riga successiva a quella iniziale e alla corrispondenza raggiunta, si ottiene la riga finale dell'intervallo;
  – se si tratta di un'espressione regolare, il confronto avviene a partire dalla riga successiva alla prima che è stata trovata.

- *riga_iniziale* , *riga_finale* !

  Se alla fine della selezione delle righe appare un punto esclamativo, questo rappresenta l'inversione della selezione, ovvero tutte le altre righe.

318.4.2 Comandi comuni

Come accennato, ogni direttiva si compone di una selezione di righe, in modo esplicito o implicito, e di un comando, ovvero di un raggruppamento di comandi racchiuso tra parentesi graffe. Vengono elencati di seguito i comandi più comuni.

- *#commento*

  Il simbolo '**#**' rappresenta un comando speciale di SED che serve solo a fargli ignorare il testo che segue fino alla fine della riga (fino alla fine della direttiva). Trattandosi di un «comando», si applica a delle righe, che però non possono essere indicate.

  Di solito, i commenti di questo tipo si inseriscono solo nei file contenenti direttive di un programma di elaborazione SED (eventualmente uno script eseguibile, realizzato nella forma che è già stata mostrata).

  > Se i primi due caratteri di un file del genere corrispondono alla stringa '**#n**', SED funziona come se fosse stata usata l'opzione '**-n**', per cui occorre fare attenzione ai commenti che appaiono nella prima riga di tali file.

- `s/`*espressione_regolare_elementare*`/`*rimpiazzo*`/`[*parametri*]

  `s`*x*`espressione_regolare_elementare`*x*`rimpiazzo`*x* [*parametri*]

  Con questo comando si vuole sostituire ciò che viene delimitato dall'espressione regolare con il testo di rimpiazzo, tenendo conto dei parametri posti eventualmente alla fine. L'espressione regolare e il testo di rimpiazzo sono delimitati e separati attraverso una barra obliqua normale, oppure da un altro simbolo scelto liberamente. Per inserire questa barra obliqua, o qualunque altro simbolo che svolga tale compito nell'espressione regolare, occorre proteggerlo con la barra obliqua inversa ('**\/**', ovvero '*****x**'**).

  L'espressione regolare può essere realizzata in modo da individuare alcune parti, delimitate attraverso '**\(**' e '**\)**' (bisogna ricordare che si tratta di espressioni regolari elementari, ovvero di BRE); in tal caso, nella stringa di rimpiazzo si può fare riferimento a questi blocchi attraverso la forma '*****n***'**, dove *n* è un numero da uno a nove, che indica l'*n*-esimo riferimento a questi raggruppamenti della parte di riga presa in considerazione dall'espressione regolare. Nella stringa di rimpiazzo si può anche utilizzare la e-commerciale ('**&**') per fare riferimento a tutto il blocco di testo a cui corrisponde l'espressione regolare stessa.

  I parametri in coda al modello, hanno il significato seguente:

  - `g`

    esegue l'operazione di rimpiazzo per tutte le corrispondenze che si possono avere sulla stessa riga, senza limitarsi alla prima soltanto;
  - `p`

    se la sostituzione ha avuto luogo, emette la riga risultante (il *pattern space*);
  - *n*

    rimpiazza solo nell'ambito dell'*n*-esima corrispondenza con l'espressione regolare;
  - `w` *file*

    se la sostituzione ha avuto luogo, scrive la riga risultante nel file indicato.

- `q`

  Termina il funzionamento di SED senza altre elaborazioni e senza leggere altro dai file in ingresso.

- `d`

  Cancella l'area di memoria dove è stata accumulata la riga letta, avviando immediatamente un ciclo nuovo.

- `p`

  Emette la riga letta, con le modifiche eventuali che gli fossero state apportate nel frattempo. È importante ricordare che questo è il comportamento predefinito di SED, a meno che venga utilizzata l'opzione '**-n**'.

  > Nella realizzazione di script per SED occorre tenere presente che alcune realizzazioni di questo emettono la riga una sola volta, anche se viene usato il comando '**p**' e non è stata usata l'opzione '**-n**', mentre altre, come nel caso di GNU, lo fanno due volte. Questi due comportamenti opposti sono ammissibili secondo lo standard POSIX.

- `n`

  Questo comando permette di passare alla prossima riga, immediatamente, tenendo conto che se non è stata usata l'opzione '**-n**', prima di passare alla prossima viene emessa quella precedente (come al solito). Lo scopo di questo comando è fare in modo che le direttive successive si trovino di fronte una riga nuova.

- `w` *file*

  Copia le righe nel file indicato, creandolo per l'occasione.

- `{` *comandi* `}`

  Un raggruppamento di comandi può essere realizzato delimitandolo tra parentesi graffe. Tuttavia, è importante osservare che in questo caso, i comandi vanno indicati ognuno in una riga differente, inoltre la parentesi graffa di chiusura deve apparire da sola in una riga. Di solito, non c'è la necessità di usare un raggruppamento, dal momento che basta ripetere la stessa selezione di righe con un altro comando.

Alcuni comandi che qui non vengono descritti, richiedono una scomposizione in più righe, indicando la continuazione attraverso il simbolo '\'. Dal momento che questi comandi non vengono mostrati, quello che si vuole far notare è che la barra obliqua inversa come simbolo si continuazione ha un significato speciale in SED, pertanto non va usata se non si conosce esattamente il risultato che si ottiene effettivamente.

318.5 Esempi

In questa sezione vengono mostrati alcuni esempi dell'utilizzo di SED. A seconda dei casi e dell'utilità della cosa, si fa riferimento a direttive fornite nella riga di comando (con o senza l'opzione '**-e**'), oppure a uno script vero e proprio.

Elaborazioni banali

```
$ sed "" prova.txt
```

Legge il file '`prova.txt`' e lo riemette tale e quale, dal momento che non è stato specificata alcuna direttiva per il programma di elaborazione.

```
$ sed -n 'p' prova.txt
```

Si ottiene lo stesso risultato dell'esempio precedente, perché prima viene usata l'opzione '**-n**' con cui si inibisce la riemissione predefinita delle righe lette, ma poi si specifica una direttiva contenente il comando '**p**' applicato a tutte le righe del flusso in ingresso.

Selezione delle righe

```
$ sed -n '1,10/p' prova.txt
```

Emette solo le prime 10 righe del file.

```
$ sed '/.\{81,\}/d' prova.txt
```

Elimina le righe più lunghe di 80 caratteri.

```
$ sed '/^$/d' prova.txt
```

Elimina tutte le righe vuote.

```
$ sed '/^---INIZIO---$/,/^---FINE---$/d' prova.txt
```

Elimina tutte le righe comprese negli intervalli delimitati da righe contenenti esclusivamente la stringa '**---INIZIO---**' e '**---FINE---**'.

```
$ sed -n '/^---INIZIO---$/,/^---FINE---$/p' prova.txt
```

Emette tutte le righe comprese negli intervalli delimitati da righe contenenti esclusivamente la stringa '**---INIZIO---**' e '**---FINE---**'.

Sostituzione del contenuto delle righe

```
$ sed 's/andato/venuto/' prova.txt
```

Sostituisce in ogni riga la prima occorrenza della stringa «andato» con la stringa «venuto».

```
$ sed 's/andato/venuto/g' prova.txt
```

Sostituisce tutte le occorrenze della stringa «andato» con la stringa «venuto».

```
$ sed 's/^/    /' prova.txt
```

Aggiunge quattro spazi all'inizio di ogni riga del file.

```
$ sed 's/\(.*\):\(.*\):\(.*\):\(.*\):\(.*\):\(.*\):\(.*\)/\1:\3/'
/etc/passwd
```

Seleziona solo il primo e il terzo campo del file '`/etc/passwd`'; in pratica, preleva il nominativo e il numero UID.

Raggruppamenti

```
#!/bin/sed -nf
{
p
w registro
}
```

L'esempio mostra l'unione di due comandi riferiti allo stesso gruppo di righe (tutte). Lo scopo è quello di emettere le righe attraverso lo standard output e di annotarle anche in un file denominato '`registro`'.

> Si osservi il fatto che la parentesi graffa di chiusura deve essere indicata da sola (come si vede nell'esempio) e di conseguenza può essere opportuno fare altrettanto per quella di apertura.

```
$ sed -n -e 'p' -e 'w registro' prova.txt
```

Questo esempio fa la stessa cosa di quello precedente, con la differenza che i comandi sono stati separati in due direttive riferite allo stesso gruppo di righe, inoltre si elaborano le righe del file '`prova.txt`'.

318.6 Riferimenti

- *Sed tutorials*

  <http://cs.smith.edu/~jfrankli/250f00/sed.html>

AWK: introduzione

AWK è un linguaggio di programmazione nato fondamentalmente per l'analisi e la rielaborazione di file di testo organizzati in una qualche forma tabellare. AWK potrebbe essere usato per fare anche di più, solo che quando si supera un certo limite di complessità, non è più conveniente il suo utilizzo.

AWK è un interprete, nel senso che i programmi fatti secondo questo linguaggio, vengono eseguiti direttamente, senza essere compilati, come nel caso degli script di shell. Un programma AWK può essere scritto in un file di testo normale, oppure può essere fornito come argomento della riga di comando dell'interprete: il binario '**awk**'. Volendo automatizzare l'avvio dell'interprete per l'esecuzione di uno script (che abbia i permessi di esecuzione opportuni), lo si può iniziare con la direttiva comune agli script di shell:

```
#!/usr/bin/awk -f
```

Nei sistemi Unix esistono diversi tipi differenti di interpreti AWK. Con GNU/Linux potrebbe essere disponibile la versione GNU ('**gawk**'), [1] che ha molte estensioni rispetto agli standard, oppure ci potrebbe essere '**mawk**'. In questo, come negli altri capitoli dedicati a AWK, si vuole fare riferimento allo standard POSIX, senza nemmeno approfondire troppo l'utilizzo di questo linguaggio.

319.1 Principio di funzionamento e struttura fondamentale

Il programma AWK tipico, è qualcosa che legge i dati provenienti da uno o più file, li analizza in qualche modo, generando un risultato che viene visualizzato direttamente o indirizzato a un altro file. Questo indica implicitamente due cose: un programma AWK non dovrebbe essere fatto per modificare i file di partenza; inoltre, si dà per scontato che ci sia una lettura dei file di origine, infatti ciò avviene di solito senza una richiesta esplicita.

Dal punto di vista di AWK, un file che viene analizzato è composto da record, corrispondenti normalmente alle righe del file di testo stesso, dove però il codice di interruzione di riga può essere specificato espressamente come qualcosa di diverso rispetto al solito.

Un programma AWK è composto fondamentalmente da *regole*, che stabiliscono il comportamento da prendere nei confronti dei dati in ingresso. I commenti sono introdotti dal simbolo '**#**' e terminano alla fine della riga; inoltre, le righe vuote e quelle bianche vengono ignorate nello stesso modo. La struttura delle regole di un programma AWK si può esprimere secondo lo schema seguente:

criterio_di_selezione { *azione* }

In pratica, ogni regola si suddivide in due parti: un'istruzione iniziale che definisce quali record prendere in considerazione e un'azione (più o meno articolata) indicata all'interno di parentesi graffe, da eseguire ogni volta che si incontra una corrispondenza con il criterio di selezione stabilito. Questa descrizione è solo una semplificazione che per il momento serve a iniziare la comprensione di questo linguaggio.[2]

Una regola di un programma AWK può contenere l'indicazione esplicita del solo criterio di selezione, o della sola azione da compiere. Ciò perché in tal caso si utilizza un'azione o un criterio di selezione predefinito (questo particolare verrà ripreso quando verranno mostrati i primi esempi).

[1] **Gawk** GNU GPL

[2] Dalla descrizione fatta, è chiaro che le parentesi graffe, indicate nello schema sintattico, fanno parte delle regole e vanno intese in senso letterale.

L'azione di una regola AWK è molto simile a un programma C, o Perl, con tante semplificazioni, dove il record selezionato viene passato attraverso dei *campi*, che ricordano i parametri delle shell comuni: '**$0**', '**$1**', '**$2**',... In pratica, un'azione di una regola AWK è un programma a sé stante, che viene eseguito ogni volta che il criterio di selezione della regola si avvera.

319.1.1 Selezione e azione predefinita

Una regola che non contenga l'indicazione del criterio di selezione, fa sì che vengano prese in considerazione tutte le righe dei dati in ingresso. In AWK, il valore booleano *Vero* si esprime con qualunque valore differente dallo zero e dalla stringa nulla, dal momento che entrambi questi rappresentano invece il valore *Falso* in un contesto booleano. In altre parole, una regola che non contenga l'indicazione del criterio di selezione, è come se avesse al suo posto il valore uno, che si traduce ogni volta in un risultato booleano *Vero*, cosa che permette la selezione di tutti i record.

Una regola che non contenga l'indicazione dell'azione da compiere, fa riferimento a un'azione predefinita, che in pratica fa sì che venga emessa attraverso lo standard output ogni riga che supera il criterio di selezione. Praticamente, è come se venisse usata l'azione '**{ print }**'. Per essere precisi, dal momento che in AWK il concetto di «predefinito» può riguardare diversi livelli, si tratta dell'azione '**{ print $0 }**'.

In pratica, se si unisse il criterio di selezione predefinito e l'azione predefinita, si avrebbe:

```
1 { print }
```

che riemette attraverso lo standard output tutti i record che legge dai file in ingresso. Bisogna ricordare però che almeno una delle due parti deve essere indicata esplicitamente: o il criterio di selezione, o l'azione.

319.1.2 Campi

Si è accennato al fatto che il testo analizzato da un programma AWK, viene visto generalmente come qualcosa composto da record suddivisi in campi. I record vengono individuati in base a un codice che li separa, corrispondente di solito al codice di interruzione di riga, per cui si ottiene l'equivalenza tra record e righe. I campi sono separati in modo analogo, attraverso un altro codice opportuno.

> Eccezionalmente, quando il codice indicato per individuare la suddivisione in campi è *<SP>*, cioè lo spazio normale, diventa indifferente la quantità di spazi utilizzati tra un campo e l'altro; inoltre, è possibile utilizzare anche i caratteri di tabulazione.

> Se per il codice che definisce la fine di un record e l'inizio di quello successivo, viene indicata la stringa nulla, ('**""**'), si intende che i record siano separati da una o più righe bianche o vuote.

Ogni record può avere un numero variabile di campi; al loro contenuto si può fare riferimento attraverso il simbolo '**$**' seguito da un numero che ne indica la posizione: '**$***n*' è il campo *n*-esimo del record attuale, ma in particolare, '**$0**' rappresenta il record completo. Il numero in questione può anche essere rappresentato da un'espressione (per esempio una variabile) che si traduce nel numero desiderato. Per esempio, se '**pippo**' è una variabile contenente il valore due, '**$pippo**' è il secondo campo.

319.1.3 Criterio di selezione e condizioni particolari

Il criterio di selezione dei record è generalmente un'espressione che viene valutata per ognuno di questi, in ordine, che, quando si avvera, permette l'esecuzione dell'azione corrispondente. Oltre a queste situazioni generali, esistono due istruzioni speciali da utilizzare come criteri di selezione: '**`BEGIN`**' e '**`END`**'. Queste due parole chiave vanno usate da sole, rappresentando rispettivamente il momento iniziale prima di cominciare la lettura dei dati in ingresso e il momento finale successivo alla lettura ed elaborazione dell'ultimo record dei dati. Le azioni che si abbinano a queste condizioni particolari servono a preparare qualcosa e a concludere un'elaborazione.

Esiste un altro caso di criterio di selezione speciale, costituito da due espressioni separate da una virgola, come si vede nello schema seguente:

espressione_1 **`,`** *espressione_2*

La prima espressione serve ad attivare il passaggio dei record; la seconda serve a disattivarlo. In pratica, quando si avvera la prima espressione, quel record e i successivi possono passare, fino a quando si avvera la seconda. Quando si avvera la seconda espressione (dopo che si era avverata la prima), il record attuale passa, ma quelli successivi non più. Se in seguito si riavvera la prima condizione, la cosa ricomincia.

Tabella 319.1. Schema complessivo dei diversi tipi di criteri di selezione in AWK.

Criterio di selezione	Descrizione
BEGIN	Esegue l'azione prima di iniziare a leggere i dati in ingresso.
END	Esegue l'azione dopo la lettura dei dati in ingresso.
espressione	Quando si avvera, esegue l'azione per il record attuale.
espr_1, *espr_2*	Le due espressioni individuano i record a intervalli.

319.1.4 Un programma banale per cominciare

Per mostrare il funzionamento di un programma AWK viene mostrato subito un esempio banale. Come è già stato descritto, la cosa più semplice che possa fare un programma AWK, è la riemissione degli stessi record letti in ingresso, senza porre limiti alla selezione.

```
1 { print $0 }
```

Come è già stato descritto, la regola mostrata è molto semplice: il numero uno rappresenta in pratica un valore corrispondente a *Vero*, dal punto di vista booleano, per cui si tratta di un'espressione che si avvera sempre, portando così alla selezione di tutti i record; l'azione richiede l'emissione della riga attuale, rappresentata da '**`$0`**'.

Se si realizza un file contenente la regola che è stata mostrata, supponendo di averlo chiamato '`banale`', per avviarlo basta il comando seguente:

```
$ awk -f banale
```

Nel comando non è stato specificato alcun file da analizzare, per cui l'interprete '**`awk`**' lo attende dallo standard input, in questo caso dalla tastiera. Per terminare la prova basta concludere l'inserimento attraverso la combinazione [*Ctrl+d*].

Un programma così breve può essere fornito direttamente nella riga di comando:

```
$ awk '1 { print $0 }'
```

Per realizzare uno script, basta mettere l'intestazione corretta al file del programma, ricordando poi di rendere eseguibile il file:

```
#!/usr/bin/awk -f
1 { print $0 }
```

Prima di proseguire, è il caso di vedere come funzionano i criteri di selezione '**BEGIN**' e '**END**':

```
BEGIN { print "Inizio del programma" }
1 { print $0 }
END { print "Fine del programma" }
```

In questo modo, prima di iniziare la riemissione del testo che proviene dal file in ingresso, viene emesso un messaggio iniziale; quindi, alla fine di tutto viene emesso un altro messaggio conclusivo.

319.1.5 Variabili predefinite

AWK ha ereditato dalle shell l'idea delle variabili predefinite, con le quali si può modificarne l'impostazione. Le variabili predefinite si distinguono dalle altre perché sono tutte espresse attraverso nomi con lettere maiuscole.

Due di queste variabili sono fondamentali: '**RS**', *Record separator*, e '**FS**', *Field separator*. La prima serve a definire il carattere da prendere in considerazione per separare i dati in ingresso in record; la seconda serve a definire il codice da prendere in considerazione per separare i record in campi. Per la precisione, nel caso della variabile '**FS**', può trattarsi di un carattere singolo, oppure di un'espressione regolare.

I valori predefiniti di queste variabili sono rispettivamente *<LF>*, ovvero il codice di interruzione di riga dei file di testo normali, e uno spazio normale, che rappresenta una situazione particolare, come è già stato descritto. Questi valori possono essere cambiati: la situazione tipica in cui si deve intervenire nella variabile '**FS**' è quella della lettura di file come '`/etc/passwd`' e simili, dove si assegna generalmente alla variabile '**FS**' il valore '`:`', che è effettivamente il carattere utilizzato per separare i campi.

319.1.6 Struttura ideale di un programma AWK

Idealmente, un programma AWK potrebbe essere rappresentato in modo più esplicito, secondo lo schema sintattico seguente, dove le parentesi graffe vanno considerate in modo letterale:

```
[function nome_funzione (parametri_formali) { istruzioni }]
…
[BEGIN { azione }]
[BEGIN { azione }]
…
espressione_di_selezione { azione }
…
[END { azione }]
[END { azione }]
…
```

L'ordine indicato non è indispensabile, tuttavia è opportuno. In pratica vengono eseguite nell'ordine le seguenti fasi:

1. vengono eseguite le azioni abbinate alle condizioni '**BEGIN**', ammesso che esistano;
2. inizia la lettura del file in ingresso;
3. per ogni record vengono valutate le espressioni di selezione;
4. per ogni espressione che si avvera, viene eseguita l'azione corrispondente (se più espressioni si avverano simultaneamente, vengono eseguite ordinatamente tutte le azioni relative);

5. alla fine, vengono eseguite le azioni abbinate alle condizioni '**END**'.

Un programma AWK potrebbe essere composto anche solo da regole di tipo '**BEGIN**' o '**END**'. Nel primo caso non è nemmeno necessario leggere i dati in ingresso, mentre nel caso ci sia una regola di tipo '**END**', ciò diventa indispensabile, perché l'azione relativa potrebbe utilizzare le informazioni generate dalla lettura stessa.

AWK mette a disposizione una serie di funzioni predefinite, consentendo la dichiarazione di altre funzioni personalizzate. L'ordine in cui appaiono queste funzioni non è importante: una funzione può richiamare anche un'altra funzione dichiarata in una posizione successiva.

319.2 Avvio dell'interprete

L'interprete di un programma AWK è l'eseguibile '**awk**', che di solito è un collegamento alla realizzazione di AWK che risulta installata effettivamente: in un sistema GNU/Linux potrebbe trattarsi di '**mawk**' o '**gawk**' (il secondo è la versione GNU di AWK). La sintassi standard di un interprete AWK dovrebbe essere quella seguente:

```
awk [-F separazione_campi] [-v variabile=valore] -f file_contenente_il_programma [--] [file_in_ingresso ...]
awk [-F separazione_campi] [-v variabile=valore] [--] 'testo_del_programma' [file_in_ingresso ...]
```

I due schemi alternativi riguardano la possibilità di far leggere all'interprete il programma contenuto in un file, indicato attraverso l'opzione '**-f**', oppure di fornirlo direttamente nella riga di comando, delimitandolo opportunamente perché venga preso dalla shell come un argomento singolo.

Se non vengono forniti i file da usare come dati in ingresso, l'interprete attende i dati dallo standard input.

Alcune opzioni

`-F` *separazione_campi*

Definisce in che modo devono essere distinti i campi dei record, modificando così il valore predefinito della variabile '**FS**'. Come è già stato descritto, può trattarsi di un carattere singolo, oppure di un'espressione regolare.

`-v` *variabile=valore*

Assegna un valore a una variabile. La variabile in questione può essere predefinita, oppure una nuova che viene utilizzata nel programma per qualche motivo.

`-f` *file_programma_awk*

Indica espressamente il file contenente il programma AWK del quale deve essere iniziata l'interpretazione.

`--`

Una coppia di trattini dichiara la conclusione delle opzioni normali e l'inizio degli argomenti finali (può essere usato per evitare ambiguità, nel caso ce ne possano essere). Gli argomenti successivi possono essere il programma stesso, se non è stata utilizzata l'opzione '**-f**', quindi i file da fornire in ingresso per l'elaborazione.

Esempi

```
$ awk -f programma.awk elenco
```

Avvia l'esecuzione del programma contenuto nel file '`programma.awk`', per l'elaborazione del file '`elenco`'.

```
$ cat elenco | awk -f programma.awk
```

Esattamente come nell'esempio precedente, con la differenza che il file '`elenco`' viene fornito attraverso lo standard input.

```
$ awk -f programma.awk -F : /etc/passwd
```

Esegue una qualche elaborazione, attraverso il programma '`programma.awk`', sui dati del file '`/etc/passwd`'. Per questo motivo, viene definito l'utilizzo del carattere '`:`' come separatore dei campi che compongono i record di quel file.

```
$ awk -f programma.awk -v FS=: /etc/passwd
```

Esattamente come nell'esempio precedente, intervenendo direttamente sulla variabile predefinita '**`FS`**'.

319.3 Espressioni

L'espressione è qualcosa che restituisce un valore. I tipi di valori gestiti da AWK sono pochi: numerici (numeri reali), stringhe e stringhe numeriche. I valori booleani non hanno un tipo indipendente: lo zero numerico e la stringa nulla valgono come *Falso*, mentre tutto il resto vale come *Vero* (anche la stringa '**`"0"`**' vale come *Vero*, a differenza di quanto accade con il linguaggio Perl).

319.3.1 Costanti

Le costanti sono espressioni elementari che restituiscono un valore in base a una simbologia convenuta. I valori numerici si esprimono in forma costante nei modi comuni anche agli altri linguaggi di programmazione. I valori interi si possono indicare come una serie di cifre numeriche, non delimitate, che esprimono il valore secondo una numerazione a base decimale; i valori non interi possono essere espressi utilizzando il punto come separatore tra la parte intera e la parte decimale; sia i valori interi che gli altri, possono essere espressi secondo la notazione esponenziale. Le costanti numeriche che appaiono di seguito, sono esempi di rappresentazione dello stesso valore: 100,5.

```
100.5
1.005e+2
1005e-1
```

Le stringhe sono delimitate da apici doppi, come si vede nell'esempio seguente:

```
"questa è una stringa"
```

Le stringhe possono contenere delle sequenze di escape, come elencato nella tabella 319.2.

Tabella 319.2. Sequenze di escape utilizzabili all'interno delle stringhe costanti.

Escape	Significato
\\	\
\"	"
\/	/
\a	<*BEL*>
\b	<*BS*>
\f	<*FF*>
\n	<*LF*>
\r	<*CR*>
\t	<*HT*>
\v	<*VT*>
nnn	il valore ottale *nnn*

AWK gestisce anche un tipo speciale di costante, che è da considerare come un tipo speciale di stringa: l'espressione regolare costante. Questa è una stringa delimitata all'inizio e alla fine da una barra obliqua normale. Per esempio,

```
/ciao/
```

è un'espressione regolare che corrisponde alla sottostringa '**ciao**'. Anche le espressioni regolari costanti ammettono l'uso di sequenze di escape e precisamente le stesse che si possono usare per le stringhe.

In generale, un'espressione regolare costante può essere usata alla destra di un'espressione di comparazione, in cui si utilizza l'operatore '**~**' o '**!~**'. Nelle altre situazioni, salvo i pochi casi in cui un'espressione regolare costante può essere indicata come parametro di una funzione, AWK sottintende che questa esprima la comparazione con il record attuale, ovvero con '**$0**'.

319.3.2 Espressioni regolari

Le espressioni regolari di AWK sono quelle estese, ovvero quelle definite da POSIX come ERE. Tuttavia, la grammatica effettiva di queste dipende dalla realizzazione dell'interprete particolare di cui si dispone. In generale dovrebbero essere disponibili gli operatori riassunti nella tabella 319.3, tenendo presente che le espressioni regolari di AWK ammettono la presenza di sequenze di escape per rappresentare caratteri che non potrebbero essere indicati altrimenti (la tabella 319.2).

Tabella 319.3. Elenco degli operatori standard delle espressioni regolari estese.

Operatore	Descrizione
\	Protegge il carattere seguente da un'interpretazione diversa da quella letterale.
^	Ancora dell'inizio di una stringa.
.	Corrisponde a un carattere qualunque.
$	Ancora della fine di una stringa.
|	Indica due possibilità alternative alla sua sinistra e alla sua destra.
( )	Definiscono un raggruppamento.
[ ]	Definiscono un'espressione tra parentesi quadre.
[*xy*...]	Un elenco di caratteri alternativi.
[*x*-*y*]	Un intervallo di caratteri alternativi.
[^...]	I caratteri che non appartengono all'insieme.
*x**	Nessuna o più volte *x*. Equivalente a '*x***{0,}**'.
x?	Nessuna o al massimo una volta *x*. Equivalente a '*x***{0,1}**'.
x+	Una o più volte *x*. Equivalente a '*x***{1,}**'.
x{*n*}	Esattamente *n* volte *x*.
x{*n*,}	Almeno *n* volte *x*.
x{*n*,*m*}	Da *n* a *m* volte *x*.

In generale, è improbabile che siano disponibili i simboli di collazione e le classi di equivalenza, come definito dallo standard POSIX per le espressioni tra parentesi quadre. Nel caso particolare della versione GNU di AWK, si possono usare le classi di caratteri (nella forma '**[:***nome***:]**'). Anche a causa di queste carenze, ogni realizzazione di AWK utilizza le proprie estensioni particolari, che di solito sono rappresentate da sequenze di escape particolari. La tabella 319.4 riepiloga le estensioni GNU, che riguardano quindi '**gawk**'.[3]

[3] Le espressioni regolari GNU prevedono normalmente la sequenza di escape '**\b**' come riferimento alla stringa nulla all'inizio o alla fine di una parola. Tuttavia, dal momento che con AWK questa sequenza deve rappresentare il carattere <*BS*> (*backspace*), allora viene sostituita dalla sequenza '**\y**'.

Tabella 319.4. Elenco delle estensioni GNU alle espressioni regolari di AWK.

Operatore	Descrizione
\y	La stringa nulla all'inizio o alla fine di una parola.
\B	La stringa nulla interna a una parola.
\<	La stringa nulla all'inizio di una parola.
\>	La stringa nulla alla fine di una parola.
\w	Un carattere di una parola, praticamente '`[[:alnum:]_]`'.
\W	L'opposto di '`\w`', praticamente '`[^[:alnum:]_]`'.

319.3.3 Campi e Variabili

Le variabili sono espressioni elementari che restituiscono il valore che contengono. AWK gestisce una serie di variabili predefinite, che possono essere lette per conoscere delle informazioni sui dati in ingresso, oppure possono essere modificate per cambiare il comportamento di AWK. Oltre a queste si possono utilizzare le variabili che si vogliono; per farlo è sufficiente assegnare loro un valore, senza bisogno di definirne il tipo.

Se in un'espressione si fa riferimento a una variabile che non è mai stata assegnata, questa restituisce la stringa nulla ('`""`'), che in un contesto numerico equivale allo zero. In questo senso, non c'è bisogno di inizializzare le variabili prima di usarle, dal momento che è noto il loro valore iniziale.

Eventualmente, una variabile può essere inizializzata a un valore determinato già al momento dell'avvio dell'interprete, attraverso l'opzione '`-v`' che è già stata descritta.

> I nomi delle variabili sono sensibili alla differenza che c'è tra la collezione alfabetica maiuscola e quella minuscola. In particolare si può osservare che, convenzionalmente, i nomi di tutte le variabili predefinite sono espressi con lettere maiuscole, mentre le variabili definite all'interno del programma tendono a essere espresse utilizzando prevalentemente lettere minuscole.

All'interno di un programma AWK, i riferimenti ai campi del record attuale si fanno attraverso la forma '`$`*n*', dove *n* rappresenta il campo *n*-esimo. Il riferimento a un campo può essere ottenuto anche utilizzando il risultato di un'espressione, quando questa è preceduta dal dollaro. In particolare, è ammissibile anche l'assegnamento di un valore a un campo, per quanto questo sia una pratica sconsigliabile, dal momento che questo fatto non ha alcun significato nei confronti dei dati originali.

319.3.4 Operazioni e operatori

Gli operatori usati per le espressioni numeriche sono più o meno gli stessi del linguaggio C. Per quanto riguarda le stringhe, è previsto il concatenamento, che si ottiene senza alcun operatore esplicito, affiancando variabili o costanti stringa. Inoltre, dovendo gestire le espressioni regolari, si aggiungono due operatori speciali per il confronto di queste con delle stringhe. La tabella 319.5 raccoglie l'elenco degli operatori disponibili in AWK.

Tabella 319.5. Riepilogo degli operatori principali utilizzabili nelle espressioni di AWK.

Operatore e operandi	Descrizione
(*espressione*)	Valuta l'espressione contenuta tra parentesi prima di analizzare la parte esterna.
++*op*	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
op++	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
--*op*	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
op--	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
+*op*	Non ha alcun effetto dal punto di vista numerico.
-*op*	Inverte il segno dell'operando numerico.
op1 + *op2*	Somma i due operandi numerici.
op1 - *op2*	Sottrae dal primo il secondo operando numerico.
op1 * *op2*	Moltiplica i due operandi numerici.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Modulo: il resto della divisione tra il primo e il secondo operando.
op1 ^ *op2*	Esponente: eleva il primo operando alla potenza del secondo.
var = *valore*	Assegna alla variabile il valore alla destra e restituisce lo stesso valore.
op1 += *op2*	*op1* = *op1* + *op2*
op1 -= *op2*	*op1* = *op1* - *op2*
op1 *= *op2*	*op1* = *op1* * *op2*
op1 /= *op2*	*op1* = *op1* / *op2*
op1 %= *op2*	*op1* = *op1* % *op2*
op1 ^= *op2*	*op1* = *op1* ^ *op2*
op1 && *op2*	AND logico, con cortocircuito.
op1 || *op2*	OR logico, con cortocircuito.
! *op*	NOT logico.
op1 > *op2*	*Vero* se il primo operando è maggiore del secondo.
op1 >= *op2*	*Vero* se il primo operando è maggiore o uguale al secondo.
op1 < *op2*	*Vero* se il primo operando è minore del secondo.
op1 <= *op2*	*Vero* se il primo operando è minore o uguale al secondo.
op1 == *op2*	*Vero* se i due operandi sono uguali.
op1 != *op2*	*Vero* se i due operandi sono diversi.
stringa ~ *regexp*	*Vero* se l'espressione regolare ha una corrispondenza con la stringa.
stringa !~ *regexp*	*Vero* se l'espressione regolare non ha alcuna corrispondenza.
stringa1 stringa2	Concatena le due stringhe.

Un tipo particolare di operatore logico è l'operatore condizionale, che permette di eseguire espressioni diverse in relazione al risultato di una condizione. La sua sintassi si esprime nel modo seguente:

```
condizione ? espressione1 : espressione2
```

In pratica, se l'espressione che rappresenta la condizione si avvera, viene eseguita la prima espressione che segue il punto interrogativo, altrimenti viene eseguita quella che segue i due punti.

Per quanto riguarda il confronto tra stringhe ed espressioni regolari, si deve tenere presente che lo scopo è solo quello di conoscere se c'è o meno una corrispondenza tra il modello e la stringa. Inoltre, è molto importante tenere in considerazione il fatto che un'espressione regolare costante, che non si trovi alla destra di un operatore '**~**', o '**!~**', viene interpretata come una forma contratta dell'espressione '**$0 ~/***regexp***/**', ovvero, si considera un confronto con il record attuale.

319.3.5 Conversione tra stringhe e numeri

Come è già stato descritto, AWK gestisce solo due tipi di dati: stringhe e numeri (reali). In base al contesto, i numeri vengono convertiti in stringhe e viceversa, solitamente in modo abbastanza trasparente. In particolare, una stringa che non possa essere interpretata come un numero, equivale a zero.

In generale, il concatenamento di stringhe, impone una trasformazione in stringa, mentre l'uso di operatori aritmetici impone una trasformazione in numero. Si osservi l'esempio:

```
uno = 1
due = 2
(uno due) + 3
```

Si tratta di tre istruzioni in sequenza, dove le prime due assegnano un valore numerico ad altrettante variabili, mentre l'ultima fa qualcosa di incredibile: concatena le due variabili, che di conseguenza vengono trattate come stringhe, generando la stringa '**"12"**'; quindi, la stringa viene riconvertita in numero, a causa dell'operatore '**+**', che richiede la somma con il numero tre. Alla fine, il risultato dell'ultima espressione è il numero 15.

La conversione da numero a stringa è banale quando si tratta di numeri interi, dal momento che il risultato è una stringa composta dalle stesse cifre numeriche che si utilizzano per rappresentare un numero intero. Al contrario, in presenza di numeri con valori decimali, entra in gioco una conversione per mezzo della funzione '**sprintf()**' (equivalente a quella del linguaggio C), che utilizza la stringa di formato contenuta nella variabile predefinita '**CONVFMT**'. Di solito, questa variabile contiene il valore '**"%.6g"**', che indica una precisione fino a sei cifre dopo la virgola, e una notazione che può essere esponenziale, oppure normale ('*intero* **.** *decimale*'), in base alla necessità. Le tabelle 319.6 e 319.7 riepilogano i simboli utilizzabili nelle stringhe di formato di '**sprintf()**'. Eventualmente, per una descrizione più dettagliata, si può leggere la pagina di manuale *sprintf(3)*.

Tabella 319.6. Elenco dei simboli utilizzabili in una stringa formattata per l'utilizzo con '**sprintf()**'.

Simbolo	Corrispondenza
%%	Segno di percentuale.
%c	Un carattere corrispondente al numero dato.
%s	Una stringa.
%d, %i	Un intero con segno in base 10.
%o	Un intero senza segno in ottale.
%x	Un intero senza segno in esadecimale.
%X	Come '**%x**', ma con l'uso di lettere maiuscole.
%e	Un numero a virgola mobile, in notazione scientifica.
%E	Come '**%e**', ma con l'uso della lettera '**E**' maiuscola.
%f	Un numero a virgola mobile, in notazione decimale fissa.
%g	Un numero a virgola mobile, secondo la notazione di '**%e**' o '**%f**'.
%G	Come '**%g**', ma con l'uso della lettera '**E**' maiuscola (se applicabile).

Tabella 319.7. Elenco dei simboli utilizzabili tra il segno di percentuale e la lettera di conversione.

Simbolo	Corrispondenza
spazio	Il prefisso di un numero positivo è uno spazio.
+	Il prefisso di un numero positivo è il segno '**+**'.
-	Allinea a sinistra rispetto al campo.
0	Utilizza zeri, invece di spazi, per allineare a destra.
#	Prefissa un numero ottale con uno zero e un numero esadecimale con 0x.
n	Un numero definisce la dimensione minima del campo.

Simbolo	Corrispondenza
.n	Per i numeri interi indica il numero minimo di cifre.
.n	Per i numeri a virgola mobile esprime la precisione, ovvero il numero di decimali.
.n	Per le stringhe definisce la lunghezza massima.

In generale, sarebbe bene non modificare il valore predefinito della variabile '**CONVFMT**', soprattutto non è il caso di ridurre la precisione della conversione, dal momento che la perdita di informazioni che ne deriverebbe, potrebbe creare anche dei gravi problemi a un programma. In altri termini, il formato di conversione condiziona la precisione dei valori che possono essere gestiti in un programma AWK.

319.3.6 Esempi di espressioni

Prima di proseguire con la descrizione del linguaggio AWK vengono mostrati alcuni esempi di programmi banali, in cui tutto si concentra sulla definizione delle espressioni per stabilire la selezione dei record. L'azione che si abbina è molto semplice: l'emissione del record selezionato attraverso l'istruzione '**print**'.

```
$ ls -l /etc | awk '$1 == "-rw-r--r--" { print $0 }'
```

L'esempio appena mostrato fornisce all'interprete AWK il programma come argomento nella riga di comando. Come si vede, il risultato del comando '**ls -l /etc**' viene incanalato attraverso una pipeline, fornendolo in ingresso al programma AWK, che si limita a selezionare i record in cui il primo campo corrisponde esattamente alla stringa '**"-rw-r--r--"**'. In pratica, vengono selezionati i record contenenti informazioni sui file che hanno solo i permessi 0644_8. L'esempio seguente ottiene lo stesso risultato, attraverso la comparazione con un'espressione regolare:

```
$ ls -l /etc | awk '$1 ~ /-rw-r--r--/ { print $0 }'
```

I due esempi successivi sono equivalenti e servono a selezionare tutti i record che non corrispondono al modello precedente.

```
$ ls -l /etc | awk '!( $1 == "-rw-r--r--" ) { print $0 }'

$ ls -l /etc | awk '!( $1 ~ /-rw-r--r--/ ) { print $0 }'
```

L'esempio seguente utilizza due espressioni, per attivare e disattivare la selezione dei record:

```
$ awk '$0 ~ /\/\*/, $0 ~ /\*\// { print $0 }' prova.c
```

In questo caso, i dati in ingresso provengono dal file '`prova.c`', che si intende essere un programma scritto in linguaggio C. Le due espressioni servono a selezionare le righe che contengono commenti nella forma '**/***...***/**'. Si osservi l'uso della barra obliqua inversa per proteggere i caratteri che altrimenti sarebbero stati interpretati diversamente.

La variante seguente è funzionalmente identica all'esempio precedente, dal momento che un'espressione regolare costante da sola, equivale a un'espressione in cui questa si paragona al record attuale.

```
$ awk '/\/\*/, /\*\// { print $0 }' prova.c
```

319.4 Istruzioni

Nel linguaggio AWK, le istruzioni possono apparire nell'ambito della dichiarazione delle azioni abbinate a un certo criterio di selezione dei record, oppure nel corpo della dichiarazione di una funzione.

Le istruzioni di AWK terminano normalmente alla fine della riga, salvo quando nella parte finale della riga appare una virgola ('`,`'), una parentesi graffa aperta ('`{`'), una doppia e-commerciale ('`&&`'), o una doppia barra verticale ('`||`'). Eventualmente, per continuare un'istruzione nella riga successiva, si può utilizzare una barra obliqua inversa esattamente alla fine della riga, come simbolo di continuazione ('`\`').

Un'istruzione può essere terminata esplicitamente con un punto e virgola finale ('`;`'), in modo da poter collocare più istruzioni in sequenza sulla stessa riga.

Come è già stato descritto, le righe vuote e quelle bianche vengono ignorate; inoltre, ciò che è preceduto dal simbolo '`#`', fino alla fine della riga, è considerato un commento.

Le istruzioni di AWK possono essere delle espressioni di assegnamento, delle chiamate di funzione, oppure delle strutture di controllo.

319.4.1 Istruzioni fondamentali

Le istruzioni fondamentali di AWK sono quelle che permettono di emettere del testo attraverso lo standard output. Si tratta di due funzioni, che però possono essere usate anche in forma di «operatori»: '**`print`**' e '**`printf`**'. La prima di queste due permette l'emissione di una o più stringhe, mentre la seconda permette di definire una stringa in base a un formato indicato, emettendone poi il risultato. In pratica, '**`printf`**' si comporta in modo analogo alla funzione omonima del linguaggio C.

`print`
`print` *espressione_1* [`,` *espressione_1*]...
`print(` *espressione_1* [`,` *espressione_1*]... `)`

Quelli che si vedono sono gli schemi sintattici della funzione (o istruzione) '**`print`**'. Se non vengono specificati degli argomenti (ovvero dei parametri), si ottiene l'emissione del testo del record attuale. Se invece vengono indicati degli argomenti, questi vengono emessi in sequenza, inserendo tra l'uno e l'altro il carattere definito dalla variabile '**`OFS`**' (*Output field separator*), che di solito corrisponde a uno spazio normale. In tutti i casi, il testo emesso da '**`print`**' termina con l'inserimento del carattere contenuto nella variabile '**`ORS`**' (*Output record separator*), che di solito corrisponde al codice di interruzione di riga.

In altri termini, nel primo caso viene emessa la stringa corrispondente al concatenamento '**`$0 ORS`**'; nel secondo e nel terzo viene emessa la stringa corrispondente al concatenamento '*espressione_1* **`OFS`** *espressione_2* **`OFS ...`** *espressione_n* **`ORS`**'.

`printf` *stringa_di_formato* `,` *espressione_1* [`,` *espressione_2*]...
`printf(` *stringa_di_formato* `,` *espressione_1* [`,` *espressione_2*]... `)`

L'istruzione, ovvero la funzione '**`printf`**', si comporta come la sua omonima del linguaggio C: il primo argomento è una stringa di formato, contenente una serie di simboli che iniziano con il simbolo '`%`', che vanno rimpiazzati ordinatamente con gli argomenti successivi. Le tabelle 319.6 e 319.7 riepilogano i simboli utilizzabili nelle stringhe di formato di '**`sprintf`**'. Eventualmente, per una descrizione più dettagliata, si può leggere la pagina di manuale *sprintf(3)*.

A differenza di '**print**', '**printf**' non fa uso delle variabili '**OFS**' e '**ORS**', dal momento che quello che serve può essere inserito tranquillamente nella stringa di formato (il carattere *<LF>*, corrispondente al codice di interruzione di riga, viene indicato con la sequenza di escape '**\n**').

319.4.2 Ridirezione dell'output

L'output generato dalle istruzioni '**print**' e '**printf**' può essere ridiretto all'interno del programma AWK stesso, utilizzando gli operatori '**>**', '**>>**' e '**|**'. Questo permette di ridirigere i dati verso file differenti; diversamente, converrebbe intervenire all'esterno del programma, per mezzo del sistema operativo.

`print` … `>` *file*
`printf` … `>` *file*
`print` … `>>` *file*
`printf` … `>>` *file*
`print` … `|` *comando*
`printf` … `|` *comando*

Utilizzando l'operatore '**>**' si ridirigono i dati verso un file, che viene azzerato inizialmente, oppure viene creato per l'occasione; con l'operatore '**>>**' si accodano dati a un file già esistente; con l'operatore '**|**' si inviano dati allo standard input di un altro comando. È importante osservare che i file e i comandi in questione, vanno indicati in una stringa. Si osservino gli esempi seguenti.

```
# annota il secondo campo nel file /tmp/prova
print $2 > "/tmp/prova"

# accoda il secondo campo nel file /tmp/prova
print $2 >> "/tmp/prova"

# definisce un comando per riordinare i dati e salvarli nel file /tmp/prova
comando = "sort > /tmp/prova"
#seleziona alcuni campi e poi invia al comando di riordino
print $2 $4 $5 | comando
```

319.4.3 Strutture di controllo di flusso

Il linguaggio AWK offre alcune strutture di controllo di flusso comuni agli altri linguaggi di programmazione. In particolare, come nel linguaggio C, è possibile raggruppare alcune istruzioni delimitandole con le parentesi graffe ('**{**…**}**').

Le strutture di controllo permettono di sottoporre l'esecuzione di una parte di codice alla verifica di una condizione, oppure permettono di eseguire dei cicli, sempre sotto il controllo di una condizione. La parte di codice che viene sottoposta a questo controllo, può essere un'istruzione singola, oppure un gruppo di istruzioni. Nel secondo caso, è necessario delimitare questo gruppo attraverso l'uso delle parentesi graffe, a cui si è appena accennato.

Dal momento che è comunque consentito di realizzare un gruppo di istruzioni che in realtà ne contiene una sola, probabilmente è meglio utilizzare sempre le parentesi graffe, in modo da evitare equivoci nella lettura del codice.[4]

La tabella 319.8 riassume la sintassi di queste strutture, la maggior parte delle quali dovrebbero essere già note dal linguaggio C, o da altri linguaggi simili.

[4] Dato che le parentesi graffe sono usate nel linguaggio AWK, se queste appaiono nei modelli sintattici indicati, queste fanno parte delle istruzioni e non della sintassi.

Tabella 319.8. Istruzioni per le strutture di controllo del flusso in AWK.

Sintassi	Descrizione
{ *istruzioni* }	Raggruppa assieme alcune istruzioni.
if (*condizione*) *istruzione* [else *istruzione*]	Struttura condizionale.
while (*condizione*) *istruzione*	Ciclo iterativo con condizione iniziale.
do *istruzione* while (*condizione*)	Ciclo iterativo con condizione alla fine.
for (*espr_1*; *espr_2*; *espr_3*) *istruzione*	Ciclo enumerativo.
break	Interrompe un ciclo iterativo o enumerativo.
continue	Riprende un ciclo iterativo o enumerativo.
exit [*espressione*]	Termina il programma restituendo il valore dell'argomento.
next	legge il prossimo record.

Data la natura di AWK, esiste un'istruzione particolare: '**`next`**'. Questa serve a passare immediatamente al record successivo.

Esempi

```
if ( $1 > 100 ) print $2
```

Se il primo campo del record attuale contiene un valore numerico superiore a 100, emette il contenuto del secondo campo.

```
if ( $1 > 100 ) {
    print $2
    contatore++
} else {
    print $3
}
```

Se il primo campo del record attuale contiene un valore numerico superiore a 100, emette il contenuto del secondo campo, incrementando la variabile '**`contatore`**' di un'unità. Altrimenti, emette solo il contenuto del terzo campo.

```
i = 1
while (i <= 10) {
    print i
    i++
}
```

Emette i numeri da 1 a 10.

```
for ( i = 1; i <= 10; i++ ) {
    print i
}
```

Esattamente come nell'esempio precedente, utilizzando un ciclo enumerativo.

```
for ( i = 1; i <= 20; i++ ) {
    if ( i != 13 ) {
        print i
    }
}
```

Emette i numeri da 1 a 20, escluso il 13.

```
for ( i = 1; i <= 20; i++ ) {
    if ( i != 13 ) {
        continue
    }
    print i
}
```

Come nell'esempio precedente, utilizzando una tecnica diversa (l'istruzione '**`continue`**' fa riprendere il ciclo prima di avere completato le altre istruzioni).

```
i = 1
while (1) {
    if ( i > 10 ) {
```

```
            break
        }
        print i
        i++
    }
```

Emette i numeri da 1 a 10, utilizzando un ciclo iterativo perpetuo (il numero 1 equivale a *Vero* per AWK), che viene interrotto dall'istruzione '**`break`**'.

319.4.4 Chiamata di funzione e funzioni predefinite

La chiamata di una funzione avviene come nel linguaggio C, tenendo conto che per evitare ambiguità, è importante mettere sempre la parentesi iniziale del gruppo dei parametri, attaccata al nome della funzione stessa:

funzione (*elenco_parametri*)

I parametri sono separati attraverso delle virgole, tenendo conto che in linea di principio si possono omettere quelli finali (si possono omettere tutti i parametri a partire da una certa posizione). I parametri che non vengono forniti sono equivalenti a stringhe nulle; in certi casi ci sono funzioni predisposte per riconoscere la mancata indicazione di tali informazioni, che così gestiscono attribuendo valori predefiniti.

Come nel linguaggio C, il passaggio dei parametri avviene per valore (salvo eccezioni), per cui i parametri in una chiamata possono essere delle espressioni più o meno articolate, che vengono valutate (senza un ordine preciso) prima della chiamata stessa.

Di seguito vengono descritte brevemente le funzioni interne (predefinite) di AWK. In particolare, le funzioni numeriche comuni sono elencate nella tabella 319.9.

Tabella 319.9. Elenco delle funzioni numeriche principali.

Funzione	Descrizione.
atan2(y, x)	Arcotangente di y/x in radianti.
cos(x)	Coseno di x espresso in radianti.
exp(x)	Funzione esponenziale (e^x).
int(x)	Parte intera di un numero reale.
log(x)	Logaritmo naturale (base e).
rand()	Numero casuale compreso tra zero e uno.
sin(x)	Seno di x espresso in radianti.
sqrt(x)	Radice quadrata di x.

`index(` *stringa*`,` *sottostringa_cercata* `)`

La funzione '**`index()`**' cerca la stringa indicata come secondo parametro nella stringa indicata come primo, cominciando da sinistra. Se trova la corrispondenza, restituisce la posizione iniziale di questa, altrimenti restituisce zero.

```
index( "Tizio", "zio" )
```

L'espressione mostrata come esempio, restituisce il valore tre, corrispondente al primo carattere in cui si ottiene la corrispondenza della stringa '**`zio`**' in '**`Tizio`**'.

`length(`[*stringa*]`)`

La funzione '**`length()`**' restituisce la lunghezza della stringa fornita come parametro, oppure, in sua mancanza, la lunghezza di '**`$0`**', ovvero del record attuale. Si osservino gli esempi.

```
length( "Tizio" )
```

Restituisce il valore cinque, dal momento che la stringa è composta da cinque caratteri.

```
length( 10 * 5 )
```

Dal momento che il parametro della funzione è un'espressione numerica, prima calcola il valore di questa espressione, ottenendo il numero 50, quindi lo trasforma in stringa e restituisce il valore due. In pratica, il numero 50 espresso in stringa è lungo due caratteri.

```
match( stringa, regexp )
```

La funzione '**match()**' cerca una corrispondenza per l'espressione regolare fornita come secondo parametro, con la stringa che appare come primo parametro. L'espressione regolare dovrebbe poter essere fornita in forma costante, senza che questo fatto venga inteso come un confronto implicito con il record attuale.

Se il confronto ha successo, viene restituita la posizione in cui inizia la corrispondenza nella stringa; inoltre, le variabili predefinite '**RSTART**' e '**RLENGTH**' vengono impostate rispettivamente a questa posizione e alla lunghezza della corrispondenza. Se il confronto fallisce, la funzione restituisce il valore zero e così viene impostata la variabile '**RSTART**', mentre '**RLENGTH**' riceve il valore `-1`.

```
sprintf( stringa_di_formato, espressione [, ...] )
```

La funzione '**sprintf()**' restituisce una stringa in base alla stringa di formato indicata come primo parametro, in cui le metavariabili '**%**...' vengono sostituite, nell'ordine, dai parametri successivi. Le metavariabili in questione sono state elencate nelle tabelle 319.6 e 319.7.

```
importo = 10000
sprintf( "Il totale è di EUR %i + IVA", importo )
```

L'espressione finale dell'esempio restituisce la stringa: «Il totale è di EUR 10000 + IVA».

```
sub( regexp, rimpiazzo [, stringa_da_modificare ] )
```

La funzione '**sub()**', cerca all'interno della stringa fornita come ultimo parametro, oppure all'interno del record attuale, la prima corrispondenza con l'espressione regolare indicata come primo parametro. Quindi, sostituisce quella corrispondenza con la stringa fornita come secondo parametro. L'espressione regolare dovrebbe poter essere fornita in forma costante, senza che questo fatto venga inteso come un confronto implicito con il record attuale.

> L'ultimo parametro deve essere una variabile, dal momento che viene passata per riferimento e il suo contenuto deve essere modificato dalla funzione.

La stringa di sostituzione (il secondo parametro), può contenere il simbolo '**&**', che in tal caso viene sostituito con la sottostringa per la quale si è avverata la corrispondenza con l'espressione regolare. Volendo inserire una e-commerciale letterale, si deve usare la sequenza '**\&**'.[5]

La funzione '**sub()**' restituisce il numero di sostituzioni eseguite, pertanto può trattarsi del valore uno o di zero.

```
frase = "ciao, come stai?"
sub( /ciao/, "salve", frase )
```

L'espressione finale dell'esempio restituisce il valore uno, dal momento che la sostituzione ha luogo, mentre la variabile '**frase**' contiene alla fine la stringa: «salve, come stai?».

```
frase = "ciao, come stai?"
sub( /ciao/, "& amico", frase )
```

[5] L'indicazione di una e-commerciale letterale può essere un problema. In generale sarebbe meglio evitarlo. In ogni caso, è necessario leggere la documentazione specifica per il tipo di interprete AWK che si utilizza, per sapere come comportarsi esattamente.

Questo esempio riutilizza la sottostringa della corrispondenza, attraverso il riferimento ottenuto con la e-commerciale. Alla fine, la variabile '**frase**' contiene: «ciao amico, come stai?».

```
gsub( regexp, rimpiazzo [, stringa_da_modificare ])
```

La funzione '**gsub()**', cerca all'interno della stringa fornita come ultimo parametro, oppure all'interno del record attuale, tutte le corrispondenze con l'espressione regolare indicata come primo parametro. Quindi, sostituisce quelle corrispondenze con la stringa fornita come secondo parametro. In pratica, si tratta di una variante di '**sub()**', in cui la sostituzione avviene in modo «globale». Valgono tutte le altre considerazioni fatte sulla funzione '**sub()**'.

```
substr( stringa, inizio [, lunghezza ])
```

La funzione '**substr()**' restituisce una sottostringa di quanto fornito come primo parametro, prendendo ciò che inizia dalla posizione del secondo parametro, per una lunghezza pari al terzo parametro, oppure, fino alla fine della stringa di partenza.

```
substr( "ciao come stai", 6, 4 )
```

L'espressione dell'esempio restituisce la stringa «come».

```
tolower( stringa )
```

La funzione '**tolower()**' restituisce la stringa fornita come parametro trasformata utilizzando solo lettere minuscole.

```
toupper( stringa )
```

La funzione '**toupper()**' restituisce la stringa fornita come parametro trasformata utilizzando solo lettere maiuscole.

319.5 Variabili predefinite

La tabella 319.10 riepiloga le variabili predefinite principali di AWK. In particolare, sono state escluse quelle che riguardano la gestione degli array.

Tabella 319.10. Elenco delle variabili predefinite principali di AWK.

Variabile	Descrizione
CONVFMT	Formato di conversione da numero a stringa.
FILENAME	Nome del file attuale in ingresso, oppure '**-**'.
FNR	Numero del record attuale nel file attuale.
FS	Separatore dei campi in lettura.
NF	Numero totale dei campi nel record attuale.
NR	Numero totale dei record letti fino a questo punto.
OFMT	Formato di emissione dei numeri (di solito si tratta di '**%.6g**').
OFS	Separatore dei campi per '**print**'.
ORS	Separatore dei record per '**print**'.
RS	Separatore dei record in lettura.
RSTART	Utilizzata da '**match()**' per annotare l'inizio di una corrispondenza.
RLENGTH	Utilizzata da '**match()**' per annotare la lunghezza di una corrispondenza.

È il caso di ribadire alcuni concetti fondamentali riferiti alle variabili '**FS**' e '**RS**'.

- I record in ingresso sono distinti in base al contenuto della variabile '**RS**'. Per restare aderenti allo standard POSIX, questa può contenere un carattere, oppure la stringa nulla. Di solito, la variabile '**RS**' contiene il carattere <*LF*>, ovvero il codice di interruzione di riga

comune nei sistemi Unix. Nel caso in cui sia indicata la stringa nulla, si è di fronte a una situazione particolare: i record sono separati da una o più righe bianche o vuote.

- I campi dei record in ingresso sono distinti in base al contenuto della variabile '**FS**'. Questa variabile può contenere un carattere singolo, oppure un'espressione regolare (senza delimitatori). La corrispondenza con il carattere, o con l'espressione regolare rappresenta ciò che viene considerato il separatore dei campi. Di solito, la variabile '**FS**' contiene il carattere *<SP>*, ovvero lo spazio, che costituisce una situazione particolare: la separazione tra i campi è ottenuta inserendo qualunque spazio orizzontale (*<SP>* o *<HT>*), di qualunque lunghezza. Questa eccezione permette di leggere agevolmente i listati tabellari in cui i dati sono incolonnati in qualche modo, attraverso spaziature più o meno ampie.

319.6 Esempi

Gli esempi che vengono mostrati qui sono molto banali e sono tratti prevalentemente da *Effective AWK Programming* di Arnold D. Robbins. Tuttavia, qui sono mostrati come script autonomi, utilizzando una notazione che potrebbe sembrare ridondante, ma che può essere utile per non confondere il principiante. Trattandosi di script autonomi, questi ricevono i dati in ingresso solo attraverso lo standard input.

```
#!/usr/bin/awk -f
1 {
    if (length($0) > max) {
        max = length($0)
    }
}
END {
    print max
}
```

Questo esempio serve a trovare la riga di lunghezza massima di un file di testo normale. In pratica, viene scandito ogni record e viene memorizzata la sua lunghezza se questa risulta superiore all'ultima misurazione effettuata. Alla fine viene emesso il contenuto della variabile che è stata usata per annotare questa informazione.

```
#!/usr/bin/awk -f
length($0) > 80 { print $0 }
```

Questo esempio emette tutte le righe di un file di testo che superano la lunghezza di 80 caratteri.

```
#!/usr/bin/awk -f
NF > 0 { print $0 }
```

In questo caso vengono emesse tutte le righe di un file di testo che hanno almeno un campo. In pratica, vengono escluse le righe bianche e quelle vuote.

```
#!/usr/bin/awk -f
1 { totale += $5 }
END { print "totale:" totale "byte" }
```

Questo programma è fatto per sommare i valori del quinto campo di ogni record. In pratica, si tratta di incanalare nel programma il risultato di un comando '**ls -l**', in modo da ottenere il totale in byte.

```
#!/usr/bin/awk -F : -f
1 { print $1 }
```

Questo programma è banale, ma ha qualcosa di speciale: la riga iniziale indica che si tratta di uno script di '`/usr/bin/awk`', che deve essere avviato con le opzioni '**-F : -f**'. In pratica, rispetto al solito, è stata aggiunta l'opzione '**-F :**', con la quale si specifica che la separazione tra i campi

dei record è data dal carattere '`:`'. Il programma, di per sé, è fatto per leggere un file composto da righe separate in questo modo, come nel caso di '`/etc/passwd`', allo scopo di emettere solo il primo campo, che, sempre nel caso si tratti di '`/etc/passwd`', corrisponde al nominativo-utente.

```
#!/usr/bin/awk -F : -f
BEGIN { print "Gli utenti seguenti accedono senza parola d'ordine:" }
$2 == "" { print $1 }
```

Si tratta di una variante dell'esempio precedente, dove si presume che i dati in ingresso provengano sicuramente dal file '`/etc/passwd`'. In questo caso, vengono visualizzati i nomi degli utenti che non hanno una parola d'ordine nel secondo campo.

```
#!/usr/bin/awk -f
END { print NR }
```

Legge il file fornito attraverso lo standard input ed emette il numero complessivo di record che lo compongono.

```
#!/usr/bin/awk -f
(NR % 2) == 0 { print }
```

In questo caso, vengono emesse solo i record pari. In pratica, l'espressione '**`(NR % 2) == 0`**' si avvera solo quando non c'è resto nella divisione della variabile '**`NR`**' per due.

319.7 Riferimenti

- Arnold D. Robbins, *Effective AWK Programming*, Free Software Foundation

  <http://www.fsf.org/manual/gawk-3.0.3/gawk.html>

  <http://www.gnu.org/manual/gawk-3.0.3/gawk.html>

AWK: funzioni e array

Un programma AWK può contenere la dichiarazione di funzioni definite liberamente. Queste dichiarazioni vanno fatte al di fuori delle regole normali. Il linguaggio AWK può gestire anche gli array, che comunque sono di tipo associativo.

320.1 Dichiarazione di funzioni

La dichiarazione di una funzione avviene in modo simile al linguaggio C, con la differenza che non si dichiara il tipo restituito dalla funzione e nemmeno quello delle variabili che ricevono i valori della chiamata.

```
function nome_della_funzione ( elenco_parametri_formali ) {
    istruzioni
}
```

La parentesi tonda aperta che introduce l'elenco dei parametri formali, **deve essere attaccata alla fine del nome della funzione che viene dichiarata**. L'elenco dei parametri formali è in pratica un elenco di nomi di variabili locali, che ricevono il valore dei parametri corrispondenti nella chiamata. Se una chiamata di funzione utilizza meno parametri di quelli che sono disponibili, le variabili corrispondenti ricevono in pratica la stringa nulla.

> È importante osservare che non è possibile dichiarare altre variabili locali, oltre a quelle che appaiono nell'elenco dei parametri formali.

```
function fattoriale(x) {
    i = x - 1
    while ( i > 0 ) {
        x *= i
        i--
    }
    return x
}
```

L'esempio mostra la dichiarazione di una funzione ricorsiva, per il calcolo del fattoriale. Si può osservare l'istruzione '**`return`**', che permette di stabilire il valore che viene restituito dalla funzione. Naturalmente sono ammissibili anche funzioni che non restituiscono un valore: queste non hanno l'istruzione '**`return`**'.

```
function somma( x, y, z, i ) {
    z = x
    for ( i = 1; i <= y; i++ ) {
        z++
    }
    return z
}
```

Un altro esempio può servire per comprendere la gestione delle variabili locali in una funzione. In questo caso si tratta di una funzione che calcola la somma dei primi due parametri che gli vengono forniti. I due parametri successivi, '**`z`**' e '**`i`**', sono dichiarati tra i parametri formali per essere usati come variabili locali; come si vede, la funzione non tiene in considerazione i valori che potrebbero trasportare.

In effetti, la funzione potrebbe utilizzare ugualmente le variabili '**`z`**' e '**`i`**', anche se queste non fossero dichiarate tra i parametri formali. In tal modo, però, queste variabili sarebbero globali, pertanto si potrebbero porre dei problemi di conflitti con altre variabili con lo stesso nome usate altrove nel programma.

```
#!/bin/awk -f
function somma( x, y, z, i ) {
    z = x
    for ( i = 1; i <= y; i++ ) {
        z++
    }
    return z
}
1 { print $1 "+" $2 "=" somma( $1, $2 ) }
```

Questo ultimo esempio mostra un programma completo per ottenere la somma dei primi due campi di ogni record fornito in ingresso.

320.2 Array

Gli array di AWK sono simili agli array associativi di Perl. A seconda dell'uso che si vuole fare di questi array, ci si può anche «dimenticare» di questa particolarità di AWK, utilizzando i soliti indici numerici, che però AWK tratta come stringhe.

320.2.1 Dichiarazione e utilizzo di un array

La dichiarazione di un array avviene nel momento in cui vi si fa riferimento. In pratica, con l'istruzione

```
a[2] = "ciao"
```

si assegna la stringa '**`"ciao"`**' all'elemento '**`"2"`**' dell'array '**`a`**'. Se l'array non esisteva già, viene creato per l'occasione. Nello stesso modo, se l'elemento '**`"2"`**' non esisteva, viene creato all'interno dell'array.

In pratica, l'array di AWK è un insieme di elementi a cui si fa riferimento con un indice libero. Il fare riferimento a un elemento che non esiste, anche solo per leggerne il contenuto, implica la creazione di tale elemento. Come si può intuire, il riferimento a un elemento che non esiste ancora, crea tale elemento assegnandogli la stringa nulla, restituendo pertanto lo stesso valore.

L'esempio seguente crea un array un po' strampalato, con una serie di valori senza un significato particolare:

```
elenco["ciao"] = "Saluti"
elenco["maramao"] = 123
elenco[3] = 345
elenco[2345] = "che bello"
```

Si intuisce che gli elementi di un array AWK non hanno un ordine preciso.

> È importante tenere presente che non è possibile riutilizzare una variabile scalare come array; nello stesso modo, non si può riutilizzare un array come se fosse una variabile scalare. Se si tenta di fare una cosa del genere, l'interprete dovrebbe bloccarsi con una segnalazione di errore.

320.2.2 Scandire gli elementi di un array

La scansione degli elementi di un array AWK può essere un problema, se si pensa alla sua natura. Per esempio, dal momento che facendo riferimento a un elemento che non esiste, lo si crea implicitamente, si capisce che non si può nemmeno andare per tentativi. Per risolvere il problema, AWK fornisce due strumenti: l'operatore '**in**' e una variante della struttura di controllo '**for**'.

Per verificare che un array contenga effettivamente l'elemento corrispondente a un certo indice, si usa l'operatore '**in**' nel modo seguente:

```
indice in array
```

Per esempio, per verificare che esista l'elemento '**prova[234]**', si può usare un'istruzione simile a quella seguente:

```
if (234 in prova) {
    print "L'elemento prova[234] corrisponde a " prova[234]
}
```

Per scandire tutti gli elementi di un array si usa la struttura di controllo '**for**' in un modo particolare:

```
for (variabile in array) istruzione
```

In pratica, per ogni elemento contenuto nell'array, viene eseguita l'istruzione (o il blocco di istruzioni) che segue, tenendo conto che alla variabile viene assegnato ogni volta l'indice dell'elemento in corso di elaborazione.

È chiaro che l'ordine in cui appaiono gli elementi dipende dall'interprete AWK; in generale dovrebbe dipendere dalla sequenza con qui questi sono stati inseriti. L'esempio seguente, scandisce un array e mostra il contenuto di ogni elemento:

```
for (i in elenco) {
    print "elenco[" i "]  " elenco[i]
}
```

320.2.3 Cancellazione di un elemento

L'eliminazione di un elemento di un array si ottiene con l'istruzione '**delete**':

```
delete array[indice]
```

Alcune realizzazioni di AWK sono in grado di eliminare completamente un array, se non si indica l'indice di un elemento. In alternativa, si ottiene questo risultato con la funzione '**split()**', come si vede sotto. L'uso di questa funzione viene mostrato più avanti.

```
split("", array)
```

Considerato che per AWK l'eliminazione di un array è precisamente l'eliminazione di tutti i suoi elementi, si potrebbe fare anche come viene mostrato nello schema seguente:

```
for (variabile in array) {
    delete array[variabile]
}
```

320.2.4 Indici numerici e indici «nulli»

Gli indici di un array AWK sono delle stringhe, quindi, se si usano dei numeri, questi vengono convertiti in stringa, utilizzando la stringa di formato contenuta nella variabile '**CONVFMT**'. Finché si usano indici numerici interi, non sorgono problemi; nel momento in cui si utilizzano valori non interi, la conversione può risentire di un troncamento, o di un'approssimazione derivata dalla conversione. In altri termini, due indici numerici differenti potrebbero puntare di fatto allo stesso elemento, perché la trasformazione in stringa li rende uguali.

L'indice di un array potrebbe essere anche una variabile mai usata prima. In tal caso, la variabile contiene la stringa nulla. Nel caso in cui questa variabile venga poi trattata in modo numerico, incrementando o decrementando il suo valore, per creare e fare riferimento a elementi dell'array che si vogliono raggiungere con indici pseudo-numerici, bisogna tenere presente che esiste anche l'elemento con indice '**""**'. Se si tenta di raggiungerlo con l'indice '**"0"**', si fallisce nell'intento.

```
1 {
    riga[n] = $0
    n++
}
END {
    for ( i=n-1; i >= 0; i-- ) {
        print riga[i]
    }
}
```

Si intuisce che il programma AWK che si vede nell'esempio serva ad accumulare tutte le righe lette nell'array '**riga**', quindi a scandire lo stesso array per emettere il testo di queste righe. Se si osserva con attenzione, di capisce che la prima riga non può essere ottenuta. Infatti, la variabile '**n**' viene utilizzata subito la prima volta, quando il suo contenuto iniziale è la stringa nulla, '**""**'; successivamente viene incrementata, facendo sì che quella stringa nulla venga intesa come uno zero, ma intanto è stato creato l'elemento '**riga[""]**'. Alla fine della lettura di tutti i record, viene scandito nuovamente l'array, trattandolo come se contenesse elementi da zero a '**n-1**'. Tuttavia, dal momento che l'elemento '**riga[0]**' non esiste, perché al suo posto c'è invece '**riga[""]**' che non viene raggiunto, si perde la prima riga.

320.2.5 Trasformare una stringa delimitata in un array

È molto importante considerare la possibilità di convertire automaticamente una stringa in un array attraverso la funzione interna '**split()**'.

```
split( stringa, array[, separatore])
```

In pratica, il primo parametro è la stringa da suddividere; il secondo è l'array da creare (nel caso esista già, vengono eliminati tutti i suoi elementi); il terzo, è il carattere, o l'espressione regolare, che si utilizza per separare gli elementi all'interno della lista. Se non viene indicato l'ultimo argomento, viene utilizzato il contenuto della variabile '**FS**' (come si può intuire). Dal momento che questo tipo di operazione è analoga alla separazione in campi di un record, anche in questo caso, se il carattere di separazione è uno spazio (<*SP*>), gli elementi vengono individuati tra delimitatori composti da sequenze indefinite di spazi e tabulazioni.

Il primo elemento dell'array creato in questo modo ha indice '**"1"**', il secondo ha indice '**"2"**', continuando così, di seguito, fino all'elemento *n*-esimo.

```
split( "uno-due-tre", elenco, "-" )
```

L'esempio che si vede crea (o ricrea) l'array '**elenco**', con tre elementi contenenti le stringhe '**uno**', '**due**' e '**tre**'. In pratica, è come se si facesse quanto segue:

```
elenco[1] = "uno"
elenco[2] = "due"
elenco[3] = "tre"
```

Se non c'è alcuna corrispondenza tra il carattere, o l'espressione regolare, che si utilizzano come ultimo argomento, viene creato solo l'elemento con indice '**"1"**', nel quale viene inserita tutta la stringa di partenza.

320.2.6 Array pseudo-multidimensionali

Gli array di AWK sono associativi, pertanto non ha senso parlare di dimensioni, in quanto è disponibile un solo indice. Tuttavia, gestendo opportunamente le stringhe, si possono individuare idealmente più dimensioni, anche se ciò non è vero nella realtà. Supponendo di voler gestire un array a due dimensioni, con indici numerici, si potrebbero indicare gli indici come nell'esempio seguente, dove si assegna un valore all'elemento ideale «1,10»:

```
elenco[1 "s" 10] = 123
```

La lettera «s» che si vede, è solo una stringa, scelta opportunamente, in modo che l'indice che si ottiene non si possa confondere con qualcosa che non si vuole. In questo caso, l'indice reale è la stringa '**1s10**'.

AWK offre un supporto a questo tipo di finzione multidimensionale. Per farlo, esiste la variabile '**SUBSEP**', che viene usata per definire il carattere di separazione. Questo carattere è generalmente *<FS>*, che si esprime in esadecimale come $1C_{16}$ e in ottale come 34_8, corrispondente per AWK alla sequenza di escape '**\034**'.

Quando si fa riferimento a un elemento di un array, in cui l'indice sia composto da una serie di valori separati con una virgola, AWK intende che questi valori debbano essere concatenati con il contenuto della variabile '**SUBSEP**'. Per esempio,

```
elenco[1, 10] = 123
```

è come se fosse stato scritto:

```
elenco[1 SUBSEP 10] = 123
```

In generale, non è opportuno modificare il valore di questa variabile, dal momento che si tratta di un carattere decisamente inusuale, allo scopo di garantire che non si possano formare degli indici uguali per elementi che dovrebbero essere differenti.

Per verificare se un elemento di un array del genere esiste, si può utilizzare lo stesso trucco:

```
(indice_1, indice_2, ...) in array
```

Parte lxv

Linguaggi macro

321 M4: introduzione 3618

321.1 Principio di funzionamento 3618
321.2 Convenzioni generali 3619
321.3 Istruzioni condizionali, iterazioni e ricorsioni 3624
321.4 Altre macro interne degne di nota 3625
321.5 Riferimenti 3628

M4: introduzione

M4 è un elaboratore di macro, nel senso che la sua elaborazione consiste nell'espandere le macro che incontra nell'input. In altri termini, si può dire che copia l'input nell'output, espandendo man mano le macro che incontra.

La logica di funzionamento di M4 è completamente diversa dai linguaggi di programmazione comuni; inoltre, le sue potenzialità richiedono molta attenzione da parte del programmatore. Detto in maniera diversa, si tratta di un linguaggio macro molto potente, ma altrettanto difficile da gestire.

L'obiettivo di questo capitolo è solo quello di mostrarne i principi di funzionamento, per permettere la comprensione, parziale, del lavoro di altri. Per citare un caso significativo, la configurazione di Sendmail (capitolo 155) viene gestita attualmente attraverso una serie di macro di M4, con le quali si genera il file '`/etc/sendmail.cf`'.

321.1 Principio di funzionamento

M4 è costituito in pratica dall'eseguibile '**`m4`**', la cui sintassi per l'avvio può essere semplificata nel modo rappresentato dallo schema seguente:

`m4` [*opzioni*] [*file_da_elaborare*]

Il file da elaborare può essere fornito come argomento, oppure attraverso lo standard input; il risultato viene emesso attraverso lo standard output e gli errori eventuali vengono segnalati attraverso lo standard error.

Per iniziare a comprendere il funzionamento di M4, si osservi il testo seguente:

```
Ciao, come stai ? dnl Che domanda!

# Questo è un commento ? dnl Sì.

Oggi è una giornata stupenda.
```

Supponendo di avere scritto questo in un file, precisamente '`prova.m4`', lo si può rielaborare con M4 in uno dei due modi seguenti (sono equivalenti).

```
$ m4 prova.m4

$ m4 < prova.m4
```

In entrambi i casi, quello che si ottiene attraverso lo standard output è il testo seguente:

```
Ciao, come stai ?
# Questo è un commento ? dnl Sì.

Oggi è una giornata stupenda.
```

Tutto ciò che M4 non riesce a interpretare come una macro rimane inalterato. Anche se il simbolo di commento è previsto e corrisponde a '`#`' (a meno che siano state usate opzioni o istruzioni particolari), i commenti non vengono eliminati: servono solo a evitare che il testo sia interpretato da M4.

L'unico commento che funzioni in modo simile a quello dei linguaggi di programmazione comuni è la macro '**`dnl`**' (è stata usata nella prima riga), con la quale viene eliminato il testo a partire da quel punto fino al codice di interruzione di riga successivo. Dal momento che viene eliminato anche il codice di interruzione di riga, si può vedere dall'esempio che la seconda riga,

quella vuota, viene inghiottita; invece, il «dnl» contenuto nella riga di commento non è stato considerato da M4.

321.2 Convenzioni generali

L'analisi di M4 sull'input viene condotta separando tutto in «elementi» (*token*), i quali possono essere classificati fondamentalmente in tre tipi: *nomi*, *stringhe* tra virgolette e caratteri singoli che non hanno significati particolari.

I nomi sono sequenze di lettere (compreso il trattino basso) e numeri, dove il primo carattere è una lettera. Una volta che M4 ha delimitato un nome, se questo viene riconosciuto come una macro, allora questa viene espansa (sostituendola al nome).

Le stringhe delimitate da virgolette richiedono l'uso di un apice di apertura e di uno di chiusura (‘**`**’ e ‘**'**’). Il risultato dell'elaborazione di una stringa di questo tipo è ciò che si ottiene eliminando il livello più esterno di apici. Per esempio:

```
``
```

corrisponde alla stringa nulla;

```
`la mia stringa'
```

corrisponde al testo ‘**la mia stringa**’;

```
``tra virgolette''
```

corrisponde a ‘**`tra virgolette'**’.

È importante tenere presente che anche i simboli usati per delimitare le stringhe possono essere modificati attraverso istruzioni di M4.

Tutto ciò che non rientra nella classificazione di nomi e stringhe delimitate tra virgolette, sono elementi sui quali non si applica alcuna trasformazione.

I commenti per M4 rappresentano solo una parte di testo che non deve essere analizzato alla ricerca di macro. Quello che si ottiene è la riproduzione di tale testo senza alcuna modifica. In linea di principio, i commenti sono delimitati dal simbolo ‘`#`’ fino alla fine della riga, cioè fino al codice di interruzione di riga. M4 permette di modificare i simboli usati per delimitare i commenti, o di annullarli del tutto.

È il caso di soffermarsi un momento su questo concetto. Quando si utilizza M4, spesso lo si fa per generare un file di configurazione o un programma scritto in un altro linguaggio. Questi tipi di file potrebbero utilizzare dei commenti, ma può essere conveniente generare nel risultato dei commenti il cui contenuto cambia in funzione di situazioni determinate. Si immagini di voler realizzare uno script di shell, in cui notoriamente il commento si introduce con lo stesso simbolo ‘`#`’, volendo comporre il commento in base a delle macro; diventa necessario fare in modo che M4 non consideri il simbolo ‘`#`’ come l'inizio di un commento.

L'unico tipo di dati che M4 può gestire sono le stringhe alfanumeriche, indipendentemente dal fatto che si usino gli apici per delimitarle. Naturalmente, una stringa contenente un numero può avere un significato particolare che dipende dal contesto.

321.2.1 Macro

M4 è un linguaggio di programmazione il cui scopo principale è quello di gestire opportunamente la sostituzione di testo in base a delle macro. Tuttavia, alcune macro potrebbero servire a ottenere qualche funzione in più rispetto alla semplice sostituzione di testo. In generale, per uniformità, si parla sempre di macro anche quando il termine potrebbe essere improprio; per la precisione si distingue tra macro interne (*builtin*), che pur non essendo dichiarate fanno parte di M4, e macro normali, dichiarate esplicitamente.

Una macro può essere «invocata» attraverso due modi possibili:

nome
nome (*parametro_1* , *parametro_2* , … *parametro_N*)

Nel primo caso si tratta di una macro senza parametri (ovvero senza argomenti); nel secondo si tratta di una macro con l'indicazione di parametri. È importante osservare che, quando si utilizzano i parametri, la parentesi aperta iniziale **deve seguire immediatamente il nome della macro** (senza spazi aggiuntivi); inoltre, se una macro non ha parametri, non si possono utilizzare le parentesi aperta e chiusa senza l'indicazione di parametri, perché questo sarebbe equivalente a fornire la stringa nulla come primo parametro.

La cosa più importante da apprendere è il modo in cui viene trattato il contenuto che appare tra parentesi, che serve a descrivere i parametri di una macro; infatti, prima di espandere la macro, viene espanso il contenuto che appare tra parentesi. Una volta espansa anche la macro con i parametri ottenuti, viene eseguita un'altra analisi del risultato, con il quale si possono eseguire altre espansioni di macro, oppure si può ottenere la semplice eliminazione delle coppie di apici dalle stringhe delimitate. Le operazioni svolte da M4 per espandere una macro sono elencate dettagliatamente di seguito.

1. Vengono suddivisi gli elementi contenuti tra parentesi ignorando gli spazi iniziali e includendo quelli finali. Per esempio,
   ```
   miamacro(a mio, d)
   ```
   è equivalente a
   ```
   miamacro(a mio,d)
   ```
2. Vengono espanse le macro contenute eventualmente tra i parametri. Continuando l'esempio precedente, supponendo che '**`mio`**' sia una macro che si espande nella stringa
   ```
   , b, c
   ```
   a causa della sostituzione di '**`mio`**', si ottiene in pratica quanto segue:
   ```
   miamacro(a , b, c,d)
   ```
   Infine, tutto si riduce a
   ```
   miamacro(a ,b,c,d)
   ```
   dove i parametri sono esattamente una '**`a`**' seguita da uno spazio e poi le altre lettere '**`b`**', '**`c`**' e '**`d`**'.
3. Una volta risolti i parametri, viene espansa la macro.
4. Il risultato dell'espansione viene rianalizzato alla ricerca di stringhe delimitate a cui togliere gli apici esterni e di altre macro da espandere.

In un certo senso si potrebbe dire che le stringhe, delimitate come previsto da M4, siano delle macro che restituiscono il contenuto in modo letterale, perdendo quindi la coppia di apici più esterni. Questo significa che ciò che appare all'interno di una tale stringa non può essere interpretato come il nome di una macro; inoltre, nemmeno i commenti vengono presi in considerazione

come tali. La differenza fondamentale rispetto alle macro normali sta nel fatto che l'espansione avviene una volta sola.

Quando si usano le stringhe delimitate tra le opzioni di una macro normale, è necessario tenere presente che queste vengono trattate la prima volta nel modo appena descritto, allo scopo di fornire i parametri effettivi alla macro, ma dopo l'espansione della macro avviene un'ulteriore elaborazione del risultato.

In generale sarebbe conveniente e opportuno indicare i parametri di una macro sempre utilizzando le stringhe delimitate, a meno di voler indicare esplicitamente altre macro. Ciò facilita la lettura umana di un linguaggio di programmazione già troppo complicato. In ogni caso, non si deve dimenticare il ruolo degli spazi finali che vengono sempre inclusi nei parametri. Per esempio, la macro '**miamacro**' mostrata sotto,

```
miamacro(‘a’ , ‘b’, ‘c’, ‘d’)
```

ha sempre come primo parametro la lettera '**a**' seguita da uno spazio; a nulla serve in questo caso l'uso degli apici, o meglio, sarebbe stato più opportuno usarli nel modo seguente:

```
miamacro(‘a ’, ‘b’, ‘c’, ‘d’)
```

È il caso di precisare che le sequenze di caratteri numerici sono comunque delle stringhe per M4, per cui '**miamacro(123)**' è perfettamente uguale a '**miamacro(‘123’)**'. Tuttavia, dal momento che un nome non può cominciare con un numero, non ci possono essere macro il cui nome corrisponda a un numero; pertanto si può evitare di utilizzare gli apici di delimitazione perché sarebbe comunque inutile.

Le stringhe delimitate, oltre che per impedire l'espansione di nomi che corrispondono a delle macro, permettono di «unire» due macro. Si osservi l'esempio seguente:

```
miamacro_x‘ciao’miamacro_y
```

l'intenzione è quella di fare rimpiazzare a M4 le macro '**miamacro_x**' e '**miamacro_y**' con qualcosa, facendo in modo che queste due parti si uniscano avendo al centro la parola «ciao». Si può intuire che non sarebbe stato possibile scrivere il testo seguente,

```
miamacro_xciaomiamacro_y
```

perché in tal modo non sarebbe stata riconosciuta alcuna macro. Secondo lo stesso principio, si può unire il risultato di due macro senza spazi aggiuntivi, utilizzando apici che delimitano una stringa nulla.

```
miamacro_x‘’miamacro_y
```

L'espansione delle macro pone un problema in più a causa del fatto che dopo l'espansione il risultato viene riletto alla ricerca di altre macro. Si osservi l'esempio seguente, supponendo che la macro '**miamacro_x**' restituisca la stringa '**miama**' nel caso in cui il suo unico parametro sia pari a '**1**'.

```
miamacro_x(1)cro_z
```

Espandendo la macro si ottiene la stringa «miama», ma dal momento che viene fatta una scansione successiva, la parola «miamacro_z» potrebbe essere un'altra macro; se fosse questo il caso, la macro verrebbe espansa a sua volta. Per evitare che accada una cosa del genere si possono usare gli apici in uno dei due modi seguenti.

```
miamacro_x(1)‘’cro_z
```

```
miamacro_x(1)‘cro_z’
```

Il problema può essere visto anche in modo opposto, se l'espansione di una macro, quando questa è attaccata a un'altra, può impedire il riconoscimento della seconda. L'esempio seguente mostra

infatti che la seconda macro, '**miamacro_y**', non può essere riconosciuta a causa dell'espansione della prima.

```
miamacro_x(1)miamacro_y
```

Una considerazione finale va fatta sulle macro che non restituiscono alcunché, ovvero che si traducono semplicemente nella stringa nulla. Spesso si tratta di macro interne che svolgono in realtà altri compiti, come potrebbe fare una funzione *void* di un linguaggio di programmazione normale. In questo senso, per una macro che non restituisce alcun valore, viene anche detto che restituisce *void*, che in questo contesto è esattamente la stringa nulla.

321.2.2 Definizione di una macro

`define(`*nome_macro* [`,` *espansione*]`)`

Come si può osservare dalla sintassi mostrata, la creazione di una macro avviene attraverso una macro interna, '**define**', per la quale deve essere fornito un parametro obbligatorio, corrispondente al nome della macro da creare, a cui si può aggiungere il valore in cui questa si deve espandere. Se non viene specificato in che modo si deve espandere la macro, si intende che si tratti della stringa nulla.

La macro '**define**' non restituisce alcun valore (a parte la stringa nulla). Si osservi l'esempio seguente:

```
1 define(‘CIAO’, ‘Ciao a tutti.’)
2 CIAO
```

Se questo file viene elaborato da M4, si ottiene il risultato seguente:

```
1
2 Ciao a tutti.
```

Come già affermato, '**define**' crea una macro ma non genera alcun risultato, pertanto viene semplicemente eliminata.

Per creare una macro che accetti delle opzioni, occorre indicare, nella stringa utilizzata per definire la sostituzione, uno o più simboli speciali. Si tratta precisamente di '**$1**', '**$2**',... '**$***n*'. Il numero massimo di parametri gestibili da M4 dipende dalla sua versione. I sistemi GNU (come GNU/Linux e GNU/Hurd) dispongono generalmente di M4 GNU [1] e questo non ha limiti particolari al riguardo, mentre le versioni presenti in altri sistemi Unix possono essere limitate a nove.

Questa simbologia richiama alla mente i parametri usati dalle shell comuni; e con la stessa analogia, il simbolo '**$0**' si espande nel nome della macro stessa.

```
1 define(‘CIAO’, ‘Ciao $1, come stai?’)
2 CIAO(‘Tizio’)
```

L'esempio è una variante di quello precedente, in cui si crea la macro '**CIAO**' che accetta un solo parametro. Il risultato dell'elaborazione del file appena mostrato è il seguente:

```
1
2 Ciao Tizio, come stai?
```

Prima di proseguire è opportuno rivedere il meccanismo dell'espansione di una macro attraverso un caso particolare. L'esempio seguente è leggermente diverso da quello precedente, in quanto vengono aggiunti gli apici attorno alla parola «come». Il risultato dell'elaborazione è però lo stesso.

```
1 define(‘CIAO’, ‘Ciao $1, ‘come’ stai?’)
2 CIAO(‘Tizio’)
```

[1] **GNU M4** GNU GPL

Infatti, quando la macro ‘**CIAO**’ viene espansa, subisce una rianalisi successiva; dal momento che viene trovata una stringa, questa viene «elaborata» restituendo semplicemente se stessa senza gli apici. Questo meccanismo ha comunque una fine, dal momento che non ci sono altre macro, per cui, l’esempio seguente,

```
1 define(‘CIAO’, ‘Ciao $1, ‘‘come’’ stai?’)
2 CIAO(‘Tizio’)
```

si traduce nel risultato:

```
1
2 Ciao Tizio, ‘come’ stai?
```

321.2.3 Simboli speciali

All’interno della stringa di definizione di una macro, oltre ai simboli ‘**$***n*’, si possono utilizzare altri codici simili, in un modo che assomiglia a quello delle shell più comuni.

‘**$#**’

Rappresenta il numero di parametri passati effettivamente a una macro. Per esempio,

```
define(‘CIAO’, ‘$#’)
CIAO
CIAO()
CIAO(primo, secondo)
```

Si traduce nel risultato seguente (si deve tenere presente che una macro chiamata con le parentesi senza alcun contenuto ha un parametro costituito dalla stringa nulla).

```
0
1
2
```

‘**$***’

Rappresenta tutti i parametri forniti effettivamente alla macro, separati da una virgola, ma soprattutto senza gli apici di delimitazione.

```
define(‘ECHO’, ‘$*’)
ECHO(uno,   due  , tre)
```

L’esempio si traduce nel modo seguente; si osservi l’effetto degli spazi prima e dopo i parametri.

```
uno,due  ,tre
```

‘**$@**’

Rappresenta tutti i parametri forniti effettivamente alla macro, separati da una virgola, con gli apici di delimitazione. La differenza rispetto a ‘**$***’ è sottile e l’esempio seguente dovrebbe permettere di comprenderne il significato.

```
define(‘CIAO’, ‘maramao’)
define(ECHO1,‘$1,$2,$3’)
define(ECHO2,‘$*’)
define(ECHO3,‘$@’)
ECHO1(CIAO,‘CIAO’,‘‘CIAO’’)
ECHO2(CIAO,‘CIAO’,‘‘CIAO’’)
ECHO3(CIAO,‘CIAO’,‘‘CIAO’’)
```

Le ultime righe del risultato che si ottiene sono le seguenti.

```
...
maramao,maramao,CIAO
maramao,maramao,CIAO
maramao,CIAO,‘CIAO’
```

321.2.4 Eliminazione di una macro

Una macro può essere eliminata attraverso la macro interna '**undefine**', secondo la sintassi seguente:

`undefine(`*nome_macro*`)`

Per esempio,

```
undefine('CIAO')
```

elimina la macro '**CIAO**', per cui, da quel punto in poi, la parola '**CIAO**' manterrà il suo valore letterale. '**undefine**' non restituisce alcun valore e può essere usata solo con un parametro, quello che rappresenta la macro che si vuole eliminare.

321.3 Istruzioni condizionali, iterazioni e ricorsioni

M4 non utilizza istruzioni vere e proprie, dal momento che tutto viene svolto attraverso delle «macro». Tuttavia, alcune macro interne permettono di gestire delle strutture di controllo.

Dal momento che il risultato dell'espansione di una macro viene scandito successivamente alla ricerca di altre macro da espandere, in qualche modo, è possibile anche la realizzazione di cicli ricorsivi. Resta il fatto che questo sia probabilmente un ottimo modo per costruire macro molto difficili da leggere e da controllare.

321.3.1 ifdef

`ifdef(`*nome_macro*`,` *stringa_se_esiste* `[,` *stringa_se_non_esiste* `])`

'**ifdef**' permette di verificare l'esistenza di una macro. Il nome di questa viene indicato come primo parametro, mentre il secondo parametro serve a definire la stringa da restituire in caso la condizione di esistenza si avveri. Se si indica il terzo parametro, questo viene restituito se la condizione di esistenza fallisce.

Esempi

```
ifdef('CIAO', '', 'define('CIAO', 'maramao')')
```

Verifica l'esistenza della macro '**CIAO**'; se questa non risulta già definita, la crea.

321.3.2 ifelse

`ifelse(`*commento*`)`

`ifelse(`*stringa_1*`,` *stringa_2*`,` *risultato_se_uguali* `[,` *risultato_se_diverse* `])`

`ifelse(`*stringa_1*`,` *stringa_2*`,` *risultato_se_uguali*`, ...` `[,` *risultato_altrimenti* `])`

La macro interna '**ifelse**' serve generalmente per confrontare una o più coppie di stringhe, restituendo un risultato se il confronto è valido o un altro risultato se il confronto fallisce.

Si tratta di una sorta di struttura di selezione ('**case**', '**switch**' e simili) in cui, ogni terna di parametri rappresenta rispettivamente le due stringhe da controllare e il risultato se queste risultano uguali. Un ultimo parametro facoltativo serve a definire un risultato da emettere nel caso l'unica o tutte le coppie da controllare non risultino uguali.

Nella tradizione di M4, è comune utilizzare '**ifelse**' con un solo parametro; in tal caso non si può ottenere alcun risultato, pertanto questo fatto viene sfruttato per delimitare un commento.

Esempi

```
ifelse(‘Questo è un commento’)
```

Utilizzando un solo parametro, ‘**ifelse**’ non restituisce alcunché.

```
ifelse(‘mio’, ‘mio’, ‘Vero’, ‘Falso’)
```

Questa istruzione restituisce la parola ‘**Vero**’.

```
ifelse(‘mio’, ‘mao’, ‘Vero’, ‘Falso’)
```

Questa istruzione restituisce la parola ‘**Falso**’.

321.3.3 shift

`shift(`*parametro* [`,` …] `)`

La macro interna ‘**shift**’ permette di eliminare il primo parametro restituendo i rimanenti separati da una virgola. La convenienza di utilizzare questa ‘**macro**’ sta probabilmente nell’uso assieme a ‘**$***’ e ‘**$#**’.

Esempi

```
shift(mio, tuo, suo)
```

Eliminando il primo parametro si ottiene il risultato seguente:

```
tuo,suo
```

321.3.4 forloop

`forloop(`*indice*`,` *inizio*`,` *fine*`,` *stringa_iterata* `)`

La macro interna ‘**forloop**’ permette di svolgere una sorta di ciclo in cui l’ultimo parametro, il quarto, viene eseguito tante volte quanto necessario a raggiungere il valore numerico espresso dal terzo parametro. Nel corso di questi cicli, il primo parametro viene trattato come una macro che di volta in volta restituisce un valore progressivo, a partire dal valore del secondo parametro, fino al raggiungimento di quello del terzo.

Esempi

```
forloop(‘i’, 1, 7, ‘i; ’)
```

Restituisce la sequenza dei numeri da uno a sette, seguiti da un punto e virgola.

```
1; 2; 3; 4; 5; 6; 7;
```

321.4 Altre macro interne degne di nota

In questa introduzione a M4 ci sono altre macro interne che è importante conoscere per comprendere le possibilità di questo linguaggio. Attraverso queste macro, descritte nelle sezioni seguenti, è possibile eliminare un codice di interruzione di riga, inserire dei file, cambiare i delimitatori dei commenti e deviare l’andamento del flusso di output.

321.4.1 dnl

```
dnl[ commento ]newline
```

‘**dnl**’ è una macro anomala nel sistema di M4: non utilizza parametri ed elimina tutto quello che appare dopo di lei fino alla fine della riga, comprendendo anche il codice di interruzione di riga. La natura di M4, in cui tutto è fatto sotto forma di macro, fa sì che ci si trovi spesso di fronte al problema di righe vuote ottenute nell’output per il solo fatto di avere utilizzato macro interne che non restituiscono alcun risultato. La macro ‘**dnl**’ serve principalmente per questo: eliminando anche il codice di interruzione di riga si risolve il problema delle righe vuote inutili.

Teoricamente, ‘**dnl**’ potrebbe essere utilizzata anche con l’aggiunta di parametri (tra parentesi). Il risultato che si ottiene è che i parametri vengono raccolti e interpretati come succederebbe con un’altra macro normale, senza però produrre risultati. Naturalmente, questo tipo di pratica è sconsigliabile.

Esempi

```
dnl Questo è un commento vero e proprio
define(‘CIAO’, ‘maramao’)dnl
CIAO
```

L’esempio mostra i due usi tipici di ‘**dnl**’: come introduzione di un commento fino alla fine della riga, oppure soltanto come un modo per sopprimere una riga che risulterebbe vuota nell’output. Il risultato dell’elaborazione è composto da una sola riga.

```
maramao
```

321.4.2 changecom

```
changecom([ simbolo_iniziale [, simbolo_finale ]])
```

‘**changecom**’ permette di modificare i simboli di apertura e di chiusura dei commenti. Solitamente, i commenti sono introdotti dal simbolo ‘**#**’ e sono terminati dal codice di interruzione di riga. Quando si utilizza M4 per produrre il sorgente di un certo linguaggio di programmazione, o un file di configurazione, è probabile che i commenti di questi file debbano essere modificati attraverso le macro stesse. In questo senso, spesso diventa utile cancellare la definizione dei commenti che impedirebbero la loro espansione.

Esempi

```
changecom(‘/*’, ‘*/’)
```

Cambia i simboli di apertura e chiusura dei commenti, facendo in modo di farli coincidere con quelli utilizzati dal linguaggio C.

```
changecom
```

Cancella la definizione dei commenti.

321.4.3 include, sinclude

```
include(file)
```

```
sinclude(file)
```

Attraverso la macro ‘**include**’ è possibile incorporare un file esterno nell’input in corso di elaborazione. Ciò permette di costruire file-macro di M4 strutturati. Tuttavia, è necessario fare attenzione alla posizione in cui si include un file esterno (si immagini un file che viene incluso nei parametri di una macro).

La differenza tra '**include**' e '**sinclude**' sta nel fatto che il secondo non segnala errori se il file non esiste.

Esempi

```
include('mio.m4')
```

Include il file 'mio.m4'.

321.4.4 divert, undivert

M4 consente l'uso di uno strano meccanismo, detto *deviazione*, o *diversion*, attraverso il quale parte del flusso dell'output può essere accantonato temporaneamente per essere rilasciato in un momento diverso. Per ottenere questo si utilizzano due macro interne: '**divert**' e '**undivert**'.

```
divert([numero_deviazione])
undivert([numero_deviazione[,...]])
```

La prima macro, '**divert**', serve ad assegnare un numero di deviazione alla parte di output generato a partire dal punto in cui questa appare nell'input. Questo numero può essere omesso e in tal caso si intende lo zero in modo predefinito.

La deviazione zero corrisponde al flusso normale; ogni altro numero positivo rappresenta una deviazione differente. Quando termina l'input da elaborare vengono rilasciati i vari blocchi accumulati di output, in ordine numerico crescente. In alternativa, si può usare la macro '**undivert**' per richiedere espressamente il recupero di output deviato; se questa viene utilizzata senza parametri, si intende il recupero di tutte le deviazioni, altrimenti si ottengono solo quelle elencate nei parametri.

Esiste un caso particolare di deviazione che serve a eliminare l'output; si ottiene utilizzando il numero di deviazione '**-1**'. Questa tecnica viene usata spesso anche come un modo per delimitare un'area di commenti che non si vuole siano riprodotti nell'output.

Come si può intuire, queste macro non restituiscono alcun valore.

Esempi

```
1 divert(1)
2 Questo testo è deviato
3 divert
4 Questo testo segue l'andamento normale
```

L'esempio si traduce nell'output seguente, dove le righe sono state numerate per facilitarne l'individuazione. Come si può notare, al termine del file di input viene rilasciato l'output deviato precedentemente.

```
1
4 Questo testo segue l'andamento normale

2 Questo testo è deviato
3
```

```
1 divert(1)
2 Questo testo è deviato
3 divert
4 Questo testo segue l'andamento normale
5 undivert(1)
```

Questo esempio è una variante di quello precedente, con la dichiarazione esplicita della richiesta di recupero dell'output deviato. Aggiungendo la macro '**undivert(1)**', si aggiunge anche un'interruzione di riga aggiuntiva (anche in questo caso vengono numerate le righe per facilitarne l'individuazione nel risultato).

```
1
4 Questo testo segue l'andamento normale
5
2 Questo testo è deviato
3
```

```
divert(-1)
Quanto qui contenuto non deve dare alcun
risultato nell'output.
Le macro generano regolarmente i loro effetti,
ma il loro output viene perduto.
divert
```

Questo esempio, mostra l'uso tipico di '**`divert(-1)`**'. Dal momento che alla fine appare la macro '**`divert`**' (senza altre righe), dall'elaborazione di questo file si ottiene solo un codice di interruzione di riga, cioè una riga vuota (quella in cui appare la macro '**`divert`**' finale).

```
divert(1)
Ciao maramao
divert(2)
Ciao Ciao
divert(-1)
undivert
```

L'uso di '**`divert(-1)`**' seguito da '**`undivert`**' permette di eliminare tutto l'output accumulato nelle varie deviazioni.

321.5 Riferimenti

La documentazione di M4 GNU, cioè quanto distribuito normalmente con i sistemi GNU, è disponibile generalmente attraverso la documentazione Info: *m4.info*.

Parte lxvi

DBMS e SQL

322 Introduzione ai DBMS 3631

322.1 Caratteristiche fondamentali 3631
322.2 Modello relazionale 3632
322.3 Gestione delle relazioni 3635
322.4 Riferimenti 3641

323 Introduzione a SQL 3642

323.1 Concetti fondamentali 3642
323.2 Tipi di dati 3642
323.3 Operatori, funzioni ed espressioni 3646
323.4 Tabelle 3648
323.5 Inserimento, eliminazione e modifica dei dati 3652
323.6 Interrogazioni di tabelle 3654
323.7 Trasferimento di dati in un'altra tabella 3661
323.8 Viste 3662
323.9 Controllare gli accessi 3662
323.10 Controllo delle transazioni 3664
323.11 Cursori 3664
323.12 Riferimenti 3666

324 PostgreSQL: struttura e preparazione 3667

324.1 Struttura dei dati nel file system 3667
324.2 Impostazione cliente-servente e amministrazione 3669
324.3 Accesso e autenticazione 3675
324.4 Configurazione nella distribuzione GNU/Linux Debian 3677
324.5 Gestione delle basi di dati 3679
324.6 Accesso a una base di dati 3680
324.7 Manutenzione delle basi di dati 3684
324.8 Maneggiare i file delle basi di dati 3685
324.9 Riferimenti 3691

325 PostgreSQL: il linguaggio 3692

325.1 Prima di iniziare 3692
325.2 Tipi di dati e rappresentazione 3692
325.3 Funzioni 3694
325.4 Esempi comuni 3695
325.5 Controllo delle transazioni 3701

325.6 Cursori 3701
325.7 Particolarità di PostgreSQL 3702
325.8 Riferimenti 3705

326 PostgreSQL: accesso attraverso PgAccess 3706

326.1 Accesso alla base di dati 3706
326.2 Gli «oggetti» secondo PgAccess 3707
326.3 Tabelle 3708
326.4 Interrogazioni e viste 3709
326.5 Stampe 3711
326.6 Riferimenti 3713

327 PostgreSQL: accesso attraverso WWW-SQL 3714

327.1 Principio di funzionamento 3714
327.2 Preparazione delle basi di dati e accesso 3714
327.3 Linguaggio di WWW-SQL 3715
327.4 Istruzioni 3720
327.5 Riferimenti 3724

328 Le funzioni e i trigger in PostgreSQL: un’esercitazione didattica 3725

328.1 Lo scenario: la base di dati «biblioteca» 3725
328.2 In classe: al lavoro! 3725
328.3 Riferimenti 3731

Indice analitico del volume 3732

Introduzione ai DBMS

Un DBMS (*Data base management system*) è, letteralmente, un sistema di gestione di *basi di dati*, che per attuare questa gestione utilizza il software. Queste «basi di dati» sono dei contenitori atti a immagazzinare una grande quantità di dati, per i quali il «sistema di gestione» è necessario a permetterne la fruizione (diverso è il problema della semplice archiviazione dei dati).

322.1 Caratteristiche fondamentali

Un DBMS, per essere considerato tale, deve avere caratteristiche determinate. Le più importanti che permettono di comprenderne il significato sono elencate di seguito.

- Un DBMS è fatto per gestire grandi quantità di dati.

  Convenzionalmente si può intendere che un gruppo di informazioni sia di grandi dimensioni quando questo non possa essere contenuto tutto simultaneamente nella memoria centrale dell'elaboratore. In generale un DBMS non dovrebbe porre limiti alle dimensioni, tranne quelle imposte dai supporti fisici in cui devono essere memorizzate le informazioni.

- I dati devono poter essere condivisibili.

  L'idea che sta alla base dei sistemi di gestione dei dati è quella di accentrare le informazioni in un sistema di amministrazione unico. In tal senso è poi necessario che questi dati siano condivisibili da diverse applicazioni e da diversi utenti.

- I dati devono essere persistenti e affidabili.

  I dati sono persistenti quando continuano a esistere dopo lo spegnimento della macchina con cui vengono elaborati; sono affidabili quando gli eventi per cui si possono produrre alterazioni accidentali sono estremamente limitati.

- L'accesso ai dati deve essere controllabile.

  Dovendo trattare una grande mole di dati in modo condiviso, è indispensabile che esistano dei sistemi di controllo degli accessi, per evitare che determinate informazioni possano essere ottenute da chi non è autorizzato, oppure che vengano modificate da chi non ne è il responsabile.

322.1.1 Livelli di astrazione dei dati

I dati gestiti da un DBMS devono essere organizzati a diversi livelli di astrazione. Generalmente si distinguono tre livelli: esterno, logico e interno.

- Il livello interno è quello usato effettivamente per la memorizzazione dei dati. In pratica è rappresentato dai file che contengono effettivamente le informazioni e dal modo con cui questi file vengono utilizzati. Il livello interno non è importante per la progettazione di una base di dati e nemmeno per la scrittura di programmi che devono interagire con il DBMS.
- Il livello logico, o concettuale, è quello che descrive i dati secondo la filosofia del DBMS particolare con cui si ha a che fare.
- Lo schema esterno è un'astrazione aggiuntiva che permette di definire dei punti di vista differenti dei dati descritti a livello logico. L'accesso ai dati avviene solo a questo livello, anche se di fatto può coincidere con il livello logico.

322.1.2 Ruoli e sigle standard

Lo studio sui DBMS ha generato degli acronimi che rappresentano persone o componenti essenziali di un sistema del genere. Questi acronimi devono essere conosciuti perché se ne fa uso abitualmente nella documentazione riferita ai DBMS.

L'organizzazione di una base di dati è compito del suo amministratore, definito DBA (*Data base administrator*). Eventualmente può trattarsi anche di più persone; in ogni caso, chi ha la responsabilità dell'amministrazione di uno o più basi di dati è un DBA.

La definizione della struttura dei dati (sia a livello logico che a livello esterno) viene fatta attraverso un linguaggio di programmazione definito DDL (*Data definition language*), la gestione dei dati avviene attraverso un altro linguaggio, detto DML (*Data manipulation language*), infine, la gestione della sicurezza viene fatta attraverso un linguaggio DCL (*Data control language*). Nella pratica, DDL, DML e DCL possono coincidere, come nel caso del linguaggio SQL.

322.2 Modello relazionale

Una base di dati può essere impostata secondo diversi tipi di modelli (logici) di rappresentazione. Quello più comune, che è anche il più semplice dal punto di vista umano, è il modello relazionale. In tal senso, un DBMS relazionale viene anche definito semplicemente come RDBMS.

Nel modello relazionale, i dati sono raccolti all'interno di *relazioni*. Ogni relazione è una raccolta di nessuna o più *tuple* di tipo omogeneo. La tupla rappresenta una singola informazione completa, in rapporto alla relazione a cui appartiene; infine, questa informazione è suddivisa in *attributi*. Una relazione, nella sua definizione, non ha una «forma» particolare, tuttavia questo concetto si presta a una rappresentazione tabellare: gli attributi sono rappresentati dalle colonne e le tuple dalle righe. Si osservi l'esempio della figura 322.1.

Figura 322.1. Relazione '**`Indirizzi(Cognome,Nome,Indirizzo,Telefono)`**'.

```
.=================================================================.
|Indirizzi                                                        |
|-----------------------------------------------------------------|
|Cognome        |Nome           |Indirizzo      |Telefono         |
|---------------|---------------|---------------|-----------------|
|Pallino        |Pinco          |Via Biglie 1   |0222,222222      |
|Tizi           |Tizio          |Via Tazi 5     |0555,555555      |
|Cai            |Caio           |Via Caini 1    |0888,888888      |
|Semproni       |Sempronio      |Via Sempi 7    |0999,999999      |
'=================================================================='
```

In una relazione, le tuple non hanno una posizione particolare, sono semplicemente contenute nell'insieme della relazione stessa. Se l'ordine ha una rilevanza per le informazioni contenute, questo elemento dovrebbe essere aggiunto tra gli attributi, senza essere determinato da un'ipotetica collocazione fisica. Osservando l'esempio, si intende che l'ordine delle righe non ha importanza per le informazioni che si vogliono trarre; al massimo, un elenco ordinato può facilitare la lettura umana, quando si è alla ricerca di un nome particolare, ma ciò non ha rilevanza nella struttura che deve avere la relazione corrispondente.

Il fatto che la posizione delle tuple all'interno della relazione non sia importante, significa che non è necessario poterle identificare: le tuple si distinguono in base al loro contenuto. In questo senso, una relazione non può contenere due tuple uguali: la presenza di doppioni non avrebbe alcun significato.

A differenza delle tuple, gli attributi devono essere identificati attraverso un nome. Infatti, il semplice contenuto delle tuple non è sufficiente a stabilire di quale attributo si tratti. Osservando la prima riga dell'esempio,

```
|Pallino       |Pinco          |Via Biglie 1   |0222,222222   |
```

diventa difficile distinguere quale sia il nome e quale il cognome. Attribuendo agli attributi (cioè alle colonne) un nome, diventa indifferente la disposizione fisica di questi all'interno delle tuple.

322.2.1 Relazioni collegate

Generalmente, una relazione da sola non è sufficiente a rappresentare tutti i dati riferiti a un problema o a un interesse della vita reale. Quando una relazione contiene tante volte le stesse informazioni, è opportuno scinderla in due o più relazioni più piccole, collegate in qualche modo attraverso dei riferimenti. Si osservi il caso delle relazioni rappresentate dalle tabelle che si vedono nella figura 322.2.

Figura 322.2. Relazioni di un'ipotetica gestione del magazzino.

```
.=========================================.
|Articoli                                 |
|-----------------------------------------|
|Codice|Descrizione    |Fornitore1|Fornitore2|
|------|---------------|----------|----------|
|vite30|Vite 3 mm      |       123|       126|
|vite40|Vite 4 mm      |       126|       127|
|dado30|Dado 3 mm      |       122|       123|
|dado40|Dado 4 mm      |       126|       127|
|rond50|Rondella 5 mm  |       123|       126|
'=========================================='

.=========================================.
|Movimenti                                |
|-----------------------------------------|
|Codice|Data      |Carico|Scarico|CodFor|CodCli|
|------|----------|------|-------|------|------|
|vite40|01/01/1999|  1200|       |   124|      |
|vite30|01/01/1999|      |    800|      |   825|
|vite30|02/01/1999|      |   1000|      |   954|
|vite30|03/01/1999|  2000|       |   127|      |
|rond50|03/01/1999|      |    500|      |   954|
'=========================================='

.=================================================.
|Fornitori                                        |
|-------------------------------------------------|
|CodFor|Ditta          |Indirizzo      |Telefono  |
|------|---------------|---------------|----------|
|   127|Vitoni spa     |Via Ferri 2    |0123,45678|
|   122|Ferroni spa    |Via Metalli 34 |0234,5678 |
|   126|Nuova Metal    |Via Industrie  |0345,6789 |
|   123|Viti e Bulloni |Via di sopra 7 |0567,9875 |
'================================================='

.=================================================.
|Clienti                                          |
|-------------------------------------------------|
|CodCli|Ditta          |Indirizzo      |Telefono  |
|------|---------------|---------------|----------|
|   925|Tendoni Max    |Via di sotto 2 |0113,44578|
|   825|Arti Plus      |Via di lato 45 |0765,23456|
'================================================='
```

La prima tabella, '**Articoli**', rappresenta l'anagrafica del magazzino di un grossista di ferramenta. Ogni articolo di magazzino viene codificato e descritto, inoltre vengono annotati i riferimenti ai codici di possibili fornitori. La seconda tabella, '**Movimenti**', elenca le operazioni di carico e di scarico degli articoli di magazzino, specificando solo il codice dell'articolo, la data, la quantità caricata o scaricata e il codice del fornitore o del cliente da cui è stato acquistato o a cui

è stato venduto l'articolo. Infine seguono le tabelle che descrivono i codici dei fornitori e quelli dei clienti.

Si può intendere che una sola tabella non avrebbe potuto essere utilizzata utilmente per esprimere tutte queste informazioni.

È importante stabilire che, nel modello relazionale, il collegamento tra le tuple delle varie relazioni avviene attraverso dei valori e non attraverso dei puntatori. Infatti, nella relazione '**`Articoli`**' l'attributo '**`Fornitore1`**' contiene il valore 123 e questo significa solo che i dati di quel fornitore sono rappresentati da quel valore. Nella relazione '**`Fornitori`**', la tupla il cui attributo '**`CodFor`**' contiene il valore 123 è quella che contiene i dati di quel particolare fornitore. Quindi, «123» non rappresenta un puntatore, ma solo una tupla che contiene quel valore nell'attributo «giusto». In questo senso si ribadisce l'indifferenza della posizione delle tuple all'interno delle relazioni.

322.2.2 Tipi di dati, domini e informazioni mancanti

Nelle relazioni, ogni attributo contiene una singola informazione elementare di un certo tipo, per il quale esiste un dominio determinato di valori possibili. Ogni attributo di ogni tupla deve contenere un valore ammissibile, nell'ambito del proprio dominio.

Spesso capitano situazioni in cui i valori di uno o più attributi di una tupla non sono disponibili per qualche motivo. In tal caso si pone il problema di attribuire a questi attributi un valore che definisca in modo non ambiguo questo stato di indeterminatezza. Questo valore viene definito come '**`NULL`**' ed è ammissibile per tutti i tipi di attributi possibili.

322.2.3 Vincoli di validità

I dati contenuti in una o più relazioni sono utili in quanto «sensati» in base al contesto a cui si riferiscono. Per esempio, considerando la relazione '**`Movimenti`**', vista precedentemente, questa deve contenere sempre un codice valido nell'attributo '**`Codice`**'. Se così non fosse, la registrazione data da quella tupla che dovesse avere un riferimento a un codice di articolo non valido, non avrebbe alcun senso, perché mancherebbe proprio l'informazione più importante: l'articolo caricato o scaricato.

Il controllo sulla validità dei dati può avvenire a diversi livelli, a seconda della circostanza. Si possono distinguere vincoli che riguardano:

1. il dominio dell'attributo stesso -- quando si tratta di definire se l'attributo può assumere il valore '**`NULL`**' o meno e quando si stabilisce l'intervallo dei valori ammissibili;
2. gli altri attributi della stessa tupla -- quando dal valore contenuto in altri attributi della stessa tupla dipende l'intervallo dei valori ammissibili per l'attributo in questione;
3. gli attributi di altre tuple -- quando dal valore contenuto negli attributi delle altre tuple della stessa relazione dipende l'intervallo dei valori ammissibili per l'attributo in questione;
4. gli attributi di tuple di altre relazioni -- quando altre relazioni condizionano la validità di un attributo determinato.

I vincoli di tupla, ovvero quelli che riguardano i primi due punti dell'elenco appena indicato, sono i più semplici da esprimere perché non occorre conoscere altre informazioni esterne alla tupla stessa. Per esempio, un attributo che esprime un prezzo potrebbe essere definito in modo tale che non sia ammissibile un valore negativo; nello stesso modo, un attributo che esprime uno sconto su un prezzo potrebbe ammettere un valore positivo diverso da zero solo se il prezzo a cui si riferisce, contenuto nella stessa tupla, supera un valore determinato.

Il caso più importante di un vincolo interno alla relazione, che coinvolge più tuple, è quello che riguarda le *chiavi*. In certe situazioni, un attributo, o un gruppo particolare di attributi di una relazione, deve essere unico per ogni tupla. Quindi, questo attributo, o questo gruppo di attributi, è valido solo quando non è già presente in un'altra tupla della stessa relazione.

Quando le informazioni sono distribuite fra più relazioni, i dati sono validi solo quando tutti i riferimenti sono validi. Volendo riprendere l'esempio della gestione di magazzino, visto precedentemente, una tupla della relazione '**`Movimenti`**' che dovesse contenere un codice di un fornitore o di un cliente inesistente, si troverebbe, in pratica, senza questa informazione.

322.2.4 Chiavi

Nella sezione precedente si è accennato alle *chiavi*. Questo concetto merita un po' di attenzione. In precedenza è stato affermato che una relazione contiene una raccolta di tuple che contano per il loro contenuto e non per la loro posizione. In questo senso non è ammissibile una relazione contenente due tuple identiche. Una *chiave* di una relazione è un gruppo di attributi che permette di identificare univocamente le tuple in essa contenute; per questo, tali attributi devono contenere dati differenti per ogni tupla.

Stabilendo quali attributi devono costituire una chiave per una certa relazione, si comprende intuitivamente che questi attributi non possono mai contenere un valore indeterminato.

Nella definizione di relazioni collegate attraverso dei riferimenti, l'oggetto di questi riferimenti deve essere una chiave per la relazione di destinazione. Diversamente non si otterrebbe un riferimento univoco a una tupla particolare.

322.3 Gestione delle relazioni

Prima di affrontare l'utilizzo pratico di una base di dati relazionale, attraverso un linguaggio di manipolazione dei dati, è opportuno considerare a livello teorico alcuni tipi di operazioni che si possono eseguire con le relazioni.

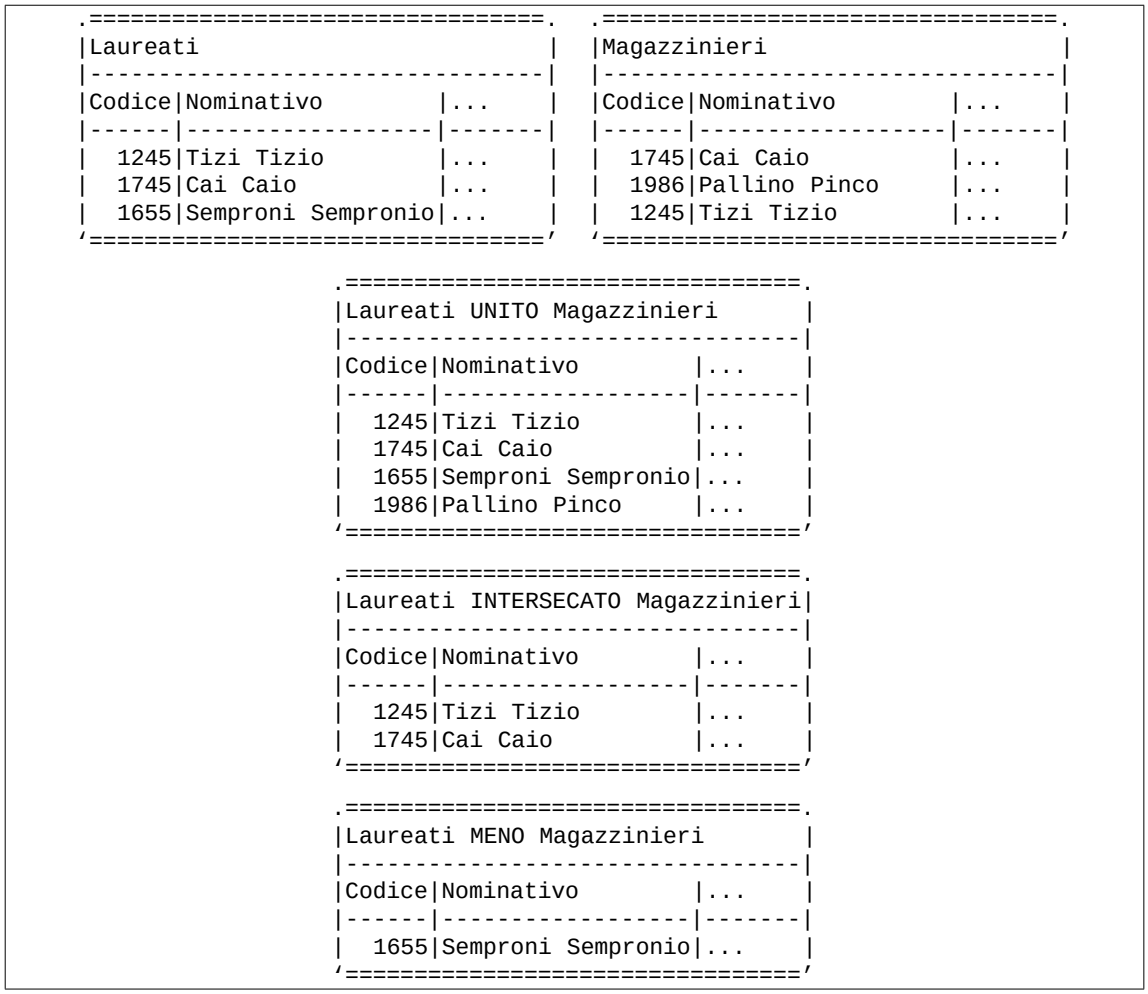
Inizialmente è stato affermato che una relazione è un insieme di tuple... Dalla teoria degli insiemi derivano molte delle operazioni che riguardano le relazioni.

322.3.1 Unione, intersezione e differenza

Quando si maneggiano relazioni contenenti gli stessi attributi, hanno senso le operazioni fondamentali sugli insiemi: unione, intersezione e differenza. Il significato è evidente: l'unione genera una relazione composta da tutte le tuple distinte delle relazioni di origine; l'intersezione genera una relazione composta dalle tuple presenti simultaneamente in tutte le relazioni di origine; la differenza genera una relazione contenente le tuple che compaiono esclusivamente nella prima delle relazioni di origine.

L'esempio rappresentato dalle tabelle della figura 322.3 dovrebbe chiarire il senso di queste affermazioni.

Figura 322.3. Unione, intersezione e differenza tra relazioni.



322.3.2 Ridenominazione degli attributi

L'elaborazione dei dati contenuti in una relazione può avvenire previa modifica dei nomi di alcuni attributi. La modifica dei nomi genera di fatto una nuova relazione temporanea, per il tempo necessario a eseguire l'elaborazione conclusiva.

Le situazioni in cui la ridenominazione degli attributi può essere conveniente possono essere varie. Nel caso delle operazioni sugli insiemi visti nella sezione precedente, la ridenominazione può rendere compatibili relazioni i cui attributi, pur essendo compatibili, hanno nomi differenti.

322.3.3 Selezione, proiezione e join

La *selezione* e la *proiezione* sono operazioni che si eseguono su una sola relazione e generano una relazione che contiene una porzione dei dati di quella di origine. La selezione permette di estrarre alcune tuple dalla relazione, mentre la proiezione estrae parte degli attribuiti di tutte le tuple. Il primo caso, quello della selezione, non richiede considerazioni particolari, mentre la proiezione ha delle implicazioni importanti.

Attraverso la proiezione, utilizzando solo parte degli attributi, si genera una relazione in cui si potrebbero perdere delle tuple, a causa della possibilità che risultino dei doppioni. Per esempio, si consideri la relazione mostrata nella figura 322.4.

Figura 322.4. Relazione '**Utenti(UID,Nominativo,Cogname,Nome,Ufficio)**'.

Utenti				
UID	Nominativo	Cognome	Nome	Ufficio
0	root	Pallino	Pinco	CED
515	rmario	Rossi	Mario	Contabilità
501	bbianco	Bianchi	Bianco	Magazzino
502	rrosso	Rossi	Rosso	Contabilità

La tabella mostra una relazione contenente le informazioni sugli utenti di un centro di elaborazione dati. Si può osservare che sia l'attributo '**UID**' che l'attributo '**Nominativo**' possono essere da soli una chiave per la relazione. Se da questa relazione si vuole ottenere una proiezione contenente solo gli attributi '**Cognome**' e '**Ufficio**', non essendo questi due una chiave della relazione, si perdono delle tuple.

Figura 322.5. Proiezione delle colonne '**Cognome**' e '**Ufficio**' della relazione '**Utenti**'.

Cognome	Ufficio
Pallino	CED
Rossi	Contabilità
Bianchi	Magazzino

La figura 322.5 mostra la proiezione della relazione '**Utenti**', in cui sono stati estratti solo gli attributi '**Cognome**' e '**Ufficio**'. In tal modo, le tuple che prima corrispondevano al numero '**UID**' 515 e 502 si sono ridotte a una sola, quella contenente il cognome «Rossi» e l'ufficio «Contabilità».

Da questo esempio si dovrebbe intendere che la proiezione ha senso, prevalentemente, quando gli attributi estratti costituiscono una chiave della relazione originaria

La congiunzione di relazioni, o *join*, è un'operazione in cui due o più relazioni vengono unite a formare una nuova relazione. Questo congiungimento implica la creazione di tuple formate dall'unione di tuple provenienti dalle relazioni di origine. Se per semplicità si pensa solo alla congiunzione di due relazioni: si va da una congiunzione minima in cui nessuna tupla della prima relazione risulta abbinata ad altre tuple della seconda, fino a un massimo in cui si ottengono tutti gli abbinamenti possibili delle tuple della prima relazione con quelle della seconda. Tra questi estremi si pone la situazione tipica, quella in cui ogni tupla della prima relazione viene collegata solo a una tupla corrispondente della seconda.

Il *join naturale* è un tipo particolare di congiunzione in cui le relazioni oggetto di tale operazione vengono collegate in base ad attributi aventi lo stesso nome. Per osservare di cosa si tratta, vale la pena di riprendere l'esempio della gestione di magazzino già descritto in precedenza. Nella figura 322.6 ne viene mostrata nuovamente solo una parte.

Figura 322.6. Le relazioni '**Articoli**' e '**Movimenti**' dell'esempio sulla gestione del magazzino.

Articoli

Codice	Descrizione	...
vite30	Vite 3 mm	...
vite40	Vite 4 mm	...
dado30	Dado 3 mm	...
dado40	Dado 4 mm	...
rond50	Rondella 5 mm	...

Movimenti

Codice	Data	Carico	Scarico	...
vite40	01/01/1999	1200		...
vite30	01/01/1999		800	...
vite30	02/01/1999		1000	...
vite30	03/01/1999	2000		...
rond50	03/01/1999		500	...

Il join naturale delle relazioni '**Movimenti**' e '**Articoli**', basato sulla coincidenza del contenuto dell'attributo '**Codice**', genera una relazione in cui appaiono tutti gli attributi delle due relazioni di origine, con l'eccezione dell'attributo '**Codice**' che appare una volta sola (figura 322.7).

Tabella 319.10. Il join naturale tra le relazioni '**Articoli**' e '**Movimenti**'.

Codice	Data	Carico	Scarico	...	Descrizione	...
vite40	01/01/1999	1200		...	Vite 4 mm	...
vite30	01/01/1999		800	...	Vite 3 mm	...
vite30	02/01/1999		1000	...	Vite 3 mm	...
vite30	03/01/1999	2000		...	Vite 3 mm	...
rond50	03/01/1999		500	...	Rondella 5 mm	...

Nel caso migliore, ogni tupla di una relazione trova una tupla corrispondente dell'altra; nel caso peggiore, nessuna tupla ha una corrispondente nell'altra relazione. L'esempio mostra che tutte le tuple della relazione '**Movimenti**' hanno trovato una corrispondenza nella relazione '**Articoli**', mentre solo alcune tuple della relazione '**Articoli**' hanno una corrispondenza dall'altra parte. Queste tuple, quelle che non hanno una corrispondenza, sono dette «penzolanti», o *dangling*, e di fatto vengono perdute dopo il join.

Quando un join genera corrispondenze per tutte le tuple delle relazioni coinvolte, si parla di join completo. La dimensione (espressa in quantità di tuple) della relazione risultante in presenza di un join completo è pari alla dimensione massima delle varie relazioni.

Quando si vuole eseguire la congiunzione di relazioni che non hanno attributi in comune si ottiene il collegamento di ogni tupla di una relazione con ogni tupla delle altre. Si può osservare l'esempio della figura 322.8 che riprende il solito problema del magazzino, con delle semplificazioni opportune.

Tabella 319.10. Le relazioni '**Articoli**' e '**Fornitori**' dell'esempio sulla gestione del magazzino.

```
.====================================.
|Articoli                            |
|------------------------------------|
|Codice|Descrizione    |Fornitore1|... |
|------|---------------|----------|----|
|vite30|Vite 3 mm      |       127|... |
|vite40|Vite 4 mm      |       127|... |
|dado30|Dado 3 mm      |       122|... |
'===================================='

     .==========================.
     |Fornitori                 |
     |--------------------------|
     |CodFor|Ditta          |... |
     |------|---------------|----|
     |   127|Vitoni spa     |... |
     |   122|Ferroni spa    |... |
     '==========================='
```

Nessuno degli attributi delle due relazioni coincide, quindi si ottiene un «prodotto» tra le due relazioni, in pratica, una relazione che contiene il prodotto delle tuple contenute nelle relazioni originali (figura 322.9).

Tabella 319.10. Il prodotto tra le relazioni '**Articoli**' e '**Fornitori**'.

```
.-------------------------------------------------------------------.
|Codice|Descrizione    |Fornitore1|... |CodFor|Ditta          |... |
|------|---------------|----------|----|------|---------------|----|
|vite30|Vite 3 mm      |       127|... |   127|Vitoni spa     |... |
|vite40|Vite 4 mm      |       127|... |   127|Vitoni spa     |... |
|dado30|Dado 3 mm      |       122|... |   127|Vitoni spa     |... |
|vite30|Vite 3 mm      |       127|... |   122|Ferroni spa    |... |
|vite40|Vite 4 mm      |       127|... |   122|Ferroni spa    |... |
|dado30|Dado 3 mm      |       122|... |   122|Ferroni spa    |... |
'==================================================================='
```

Quando si esegue un'operazione del genere, è normale che molte delle tuple risultanti siano prive di significato per gli scopi che ci si prefigge. Di conseguenza, quasi sempre, si applica poi una selezione attraverso delle condizioni. Nel caso dell'esempio, sarebbe ragionevole porre come condizione di selezione l'uguaglianza tra i valori dell'attributo '**Fornitore1**' e '**CodFor**'.

Tabella 319.10. La selezione delle tuple che rispettano la condizione di uguaglianza tra gli attributi '**Fornitore1**' e '**CodFor**'.

```
Fornitore1 = CodFor
.-------------------------------------------------------------------.
|Codice|Descrizione    |Fornitore1|... |CodFor|Ditta          |... |
|------|---------------|----------|----|------|---------------|----|
|vite30|Vite 3 mm      |       127|... |   127|Vitoni spa     |... |
|vite40|Vite 4 mm      |       127|... |   127|Vitoni spa     |... |
|dado30|Dado 3 mm      |       122|... |   122|Ferroni spa    |... |
'==================================================================='
```

Generalmente, nella pratica, non esiste la possibilità di definire un join basato sull'uguaglianza dei nomi degli attributi. Di solito si esegue un join che genera un prodotto tra le relazioni, quindi si applicano delle condizioni di selezione come nell'esempio mostrato. Quando la selezione in questione è del tipo visto nell'esempio, cioè basata sull'uguaglianza del contenuto di attributi delle diverse relazioni (anche se il nome di questi attributi è differente), si parla di *equi-join*.

322.3.4 Gestione dei valori nulli

Si è accennato in precedenza alla possibilità che gli attributi di una relazione possano contenere anche il valore indeterminato, o '**NULL**'. Con questo valore indeterminato non si possono fare comparazioni con valori determinati e di solito nemmeno con altri valori indeterminati. Per esempio, non si può dire che '**NULL**' sia maggiore o minore di qualcosa; una comparazione del genere genera solo un risultato indeterminato. '**NULL**' è solo uguale a se stesso ed è diverso da ogni altro valore, compreso un altro valore '**NULL**'.

Per verificare la presenza o l'assenza di un valore indeterminato si utilizzano generalmente operatori specifici, come in SQL:

- '**IS NULL**' -- che si avvera quando il valore controllato è indeterminato;
- '**IS NOT NULL**' -- che si avvera quando il valore controllato è determinato, quindi diverso da indeterminato.

Nel momento in cui si eseguono delle espressioni logiche, utilizzando i soliti operatori AND, OR e NOT, si pone il problema di stabilire cosa accade quando si presentano valori indeterminati. La soluzione è intuitiva: quando non si può fare a meno di conoscere il valore che si presenta come indeterminato, il risultato è indeterminato. Questo concetto deriva dalla cosiddetta logica *fuzzy*.

Figura 322.11. Tabella della verità degli operatori AND e OR quando sono coinvolti valori indefiniti.

```
? AND ? = ?     ? OR ? = ?
? AND F = F     ? OR F = ?
? AND T = ?     ? OR T = T
F AND ? = F     F OR ? = ?
F AND F = F     F OR F = F
F AND T = F     F OR T = T
T AND ? = ?     T OR ? = T
T AND F = F     T OR F = T
T AND T = T     T OR T = T
```

322.3.5 Relazioni derivate e viste

Precedentemente si è accennato al fatto che la rappresentazione finale dei dati può essere diversa da quella logica. Nel modello relazionale è possibile ottenere delle relazioni derivate da altre, attraverso una funzione determinata che stabilisce il modo con cui ottenere queste derivazioni. Si distingue fondamentalmente tra:

- relazioni derivate virtuali, o *viste*, che non generano nuove relazioni memorizzate nella base di dati, il cui contenuto viene generato al volo al momento della necessità;
- relazioni derivate materializzate, che generano una nuova relazione nella base di dati.

Il primo dei due casi è semplice da gestire, perché i dati sono sempre allineati correttamente, ma è pesante dal punto di vista elaborativo; il secondo ha invece i pregi e i difetti opposti. Con il termine «vista» si intende fare riferimento alle relazioni derivate virtuali.

322.4 Riferimenti

- Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Riccardo Torlone, *Basi di dati, concetti, linguaggi e architetture*, McGraw-Hill

Introduzione a SQL

SQL è l'acronimo di *Structured query language* e identifica un linguaggio di interrogazione (gestione) per basi di dati relazionali. Le sue origini risalgono alla fine degli anni 1970 e questo giustifica la sua sintassi prolissa e verbale tipica dei linguaggi dell'epoca, come il COBOL.

Allo stato attuale, data la sua evoluzione e standardizzazione, l'SQL rappresenta un riferimento fondamentale per la gestione di una base di dati relazionale.

A parte il significato originale dell'acronimo, SQL è un linguaggio completo per la gestione di una base di dati relazionale, includendo le funzionalità di un DDL (*Data description language*), di un DML (*Data manipulation language*) e di un DCL (*Data control language*).

Data l'età e la conseguente evoluzione di questo linguaggio, si sono definiti nel tempo diversi livelli di standard. I più importanti sono SQL89 e SQL92, noti anche come SQL2 e SQL3. Il livello SQL3 è ancora in corso di definizione.

L'aderenza dei vari sistemi DBMS allo standard SQL2 non è mai completa e perfetta, per questo sono stati definiti dei sottolivelli di questo standard per definire il grado di compatibilità di un DBMS. Si tratta di: *entry SQL*, *intermediate SQL* e *full SQL*. Si può intendere che il primo sia il livello di compatibilità minima e l'ultimo rappresenti la compatibilità totale. Lo standard di fatto è rappresentato prevalentemente dal primo livello, che coincide fondamentalmente con lo standard precedente, SQL89.

323.1 Concetti fondamentali

Convenzionalmente, le istruzioni di questo linguaggio sono scritte con tutte le lettere maiuscole. Si tratta solo di una tradizione di quell'epoca. SQL non distingue tra lettere minuscole e maiuscole nelle parole chiave delle istruzioni e nemmeno nei nomi di tabelle, colonne e altri oggetti. Solo quando si tratta di definire il contenuto di una variabile, allora le differenze contano.

In questo capitolo e nel resto del documento, quando si fa riferimento a istruzioni SQL, queste vengono indicate utilizzando solo lettere maiuscole, come richiede la tradizione.

I nomi degli oggetti (tabelle e altro) possono essere composti utilizzando lettere, numeri e il trattino basso; il primo carattere deve essere una lettera oppure il trattino basso.

Le istruzioni SQL possono essere distribuite su più righe, senza una regola precisa. Si distingue la fine di un'istruzione dall'inizio di un'altra attraverso la presenza di almeno una riga vuota. Alcuni sistemi SQL richiedono l'uso di un simbolo di terminazione delle righe, che potrebbe essere un punto e virgola.

L'SQL standard prevede la possibilità di inserire commenti; per questo si può usare un trattino doppio (‘`--`’) seguito dal commento desiderato, fino alla fine della riga.

323.2 Tipi di dati

I tipi di dati gestibili con il linguaggio SQL sono molti. Fondamentalmente si possono distinguere tipi contenenti: valori numerici, stringhe e informazioni data-orario. Nelle sezioni seguenti vengono descritti solo alcuni dei tipi definiti dallo standard.

323.2.1 Stringhe di caratteri

Si distinguono due tipi di stringhe di caratteri in SQL: quelle a dimensione fissa, completate a destra dal carattere spazio, e quelle a dimensione variabile.

`CHARACTER` | `CHARACTER(` *dimensione* `)`
`CHAR` | `CHAR(` *dimensione* `)`

Quelle appena mostrate sono le varie sintassi alternative che possono essere utilizzate per definire una stringa di dimensione fissa. Se non viene indicata la dimensione tra parentesi, si intende una stringa di un solo carattere.

`CHARACTER VARYING(` *dimensione* `)`
`CHAR VARYING(` *dimensione* `)`
`VARCHAR(` *dimensione* `)`

Una stringa di dimensione variabile può essere definita attraverso uno dei tre modi appena elencati. È necessario specificare la dimensione massima che questa stringa potrà avere. Il minimo è rappresentato dalla stringa nulla.

323.2.1.1 Costanti stringa

Le costanti stringa si esprimono delimitandole attraverso apici singoli, oppure apici doppi, come nell'esempio seguente:

```
'Questa è una stringa letterale per SQL'
"Anche questa è una stringa letterale per SQL"
```

Non tutti i sistemi SQL accettano entrambi i tipi di delimitatori di stringa. In caso di dubbio è bene limitarsi all'uso degli apici singoli; eventualmente, per inserire un apice singolo in una stringa delimitata con apici singoli, dovrebbe essere sufficiente il suo raddoppio. In pratica, per scrivere una stringa del tipo «l'albero», dovrebbe essere possibile scrivere:

```
'l''albero'
```

323.2.2 Valori numerici

I tipi numerici si distinguono in *esatti* e *approssimati*, intendendo con la prima definizione quelli di cui si conosce il numero massimo di cifre numeriche intere e decimali, mentre con la seconda si fa riferimento ai tipi a virgola mobile. In ogni caso, le dimensioni massime o la precisione massima che possono avere tali valori dipende dal sistema in cui vengono utilizzati.

`NUMERIC` | `NUMERIC(` *precisione* [`,` *scala*] `)`

Il tipo '**`NUMERIC`**' permette di definire un valore numerico composto da un massimo di tante cifre numeriche quante indicate dalla precisione, cioè il primo argomento tra parentesi. Se viene specificata anche la scala, si intende riservare quella parte di cifre per quanto appare dopo la virgola. Per esempio, con '**`NUMERIC(5,2)`**' si possono rappresentare valori da +999,99 a -999,99.

Se non viene specificata la scala, si intende che si tratti solo di valori interi; se non viene specificata nemmeno la precisione, viene usata la definizione predefinita per questo tipo di dati, che dipende dalle caratteristiche del DBMS.

`DECIMAL` | `DECIMAL(` *precisione* [`,` *scala*] `)`
`DEC` | `DEC(` *precisione* [`,` *scala*] `)`

Il tipo '**DECIMAL**' è simile al tipo '**NUMERIC**', con la differenza che le caratteristiche della precisione e della scala rappresentano le esigenze minime, mentre il sistema potrà fornire una rappresentazione con precisione o scala maggiore.

`INTEGER` | `INT`
`SMALLINT`

I tipi '**INTEGER**' e '**SMALLINT**' rappresentano tipi interi la cui dimensione dipende generalmente dalle caratteristiche del sistema operativo e dall'hardware utilizzato. L'unico riferimento sicuro è che il tipo '**SMALLINT**' permette di rappresentare interi con una precisione inferiore o uguale al tipo '**INTEGER**'.

`FLOAT` | `FLOAT(`*precisione*`)`
`REAL`
`DOUBLE PRECISION`

Il tipo '**FLOAT**' definisce un tipo numerico approssimato (a virgola mobile) con una precisione binaria pari o superiore di quella indicata tra parentesi (se non viene indicata, dipende dal sistema).

Il tipo '**REAL**' e il tipo '**DOUBLE PRECISION**' sono due tipi a virgola mobile con una precisione prestabilita. Questa precisione dipende dal sistema, ma in generale, il secondo dei due tipi deve essere più preciso dell'altro.

323.2.2.1 Costanti numeriche

I valori numerici costanti vengono espressi attraverso la semplice indicazione del numero senza delimitatori. La virgola di separazione della parte intera da quella decimale si esprime attraverso il punto ('**.**').

323.2.3 Valori Data-orario e intervalli di tempo

I valori data-orario sono di tre tipi e servono rispettivamente a memorizzare un giorno particolare, un orario normale e un'informazione data-ora completa.

`DATE`
`TIME` | `TIME(`*precisione*`)`
`TIME WITH TIME ZONE` | `TIME(`*precisione*`) WITH TIME ZONE`
`TIMESTAMP` | `TIMESTAMP(`*precisione*`)`
`TIMESTAMP WITH TIME ZONE` | `TIMESTAMP(`*precisione*`) WITH TIME ZONE`

Il tipo '**DATE**' permette di rappresentare delle date composte dall'informazione anno-mese-giorno. Il tipo '**TIME**' permette di rappresentare un orario particolare, composto da ore-minuti-secondi ed eventualmente frazioni di secondo. Se viene specificata la precisione, si intende definire un numero di cifre per la parte frazionaria dei secondi, altrimenti si intende che non debbano essere memorizzate le frazioni di secondo. Il tipo '**TIMESTAMP**' è un'informazione oraria più completa del tipo '**TIME**' in quanto prevede tutte le informazioni, dall'anno ai secondi, oltre alle eventuali frazioni di secondo. Se viene specificata la precisione, si intende definire un numero di cifre per la parte frazionaria dei secondi, altrimenti si intende che non debbano essere memorizzate le frazioni di secondo.

L'aggiunta dell'opzione '**WITH TIME ZONE**' serve a specificare un tipo orario differente, che assieme all'informazione oraria aggiunge lo scostamento, espresso in ore e minuti, dell'ora locale

dal tempo universale (UTC). Per esempio, 22:05:10+1:00 rappresenta le 22.05 e 10 secondi dell'ora locale italiana (durante l'inverno), mentre il tempo universale corrispondente sarebbe invece 21:05:10+0:00.

Quanto mostrato fino a questo punto, rappresenta un valore che indica un momento preciso nel tempo: una data o un'orario, o entrambe le cose. Per rappresentare una durata, si parla di intervalli. Per l'SQL si possono gestire gli intervalli a due livelli di precisione: anni e mesi; oppure giorni, ore, minuti, secondi ed eventualmente anche le frazioni di secondo. L'intervallo si indica con la parola chiave '**`INTERVAL`**', seguita eventualmente dalla precisione con qui questo deve essere rappresentato:

`INTERVAL` [*unità_di_misura_data_orario* [`TO` *unità_di_misura_data_orario*]]

In pratica, si può indicare che si tratta di un intervallo, senza specificare altro, oppure si possono definire una o due unità di misura che limitano la precisione di questo (pur restando nei limiti a cui si è già accennato). Tanto per fare un esempio concreto, volendo definire un'intervallo che possa esprimere solo ore e minuti, si potrebbe dichiarare con: '**`INTERVAL HOUR TO MINUTE`**'. La tabella 323.1 elenca le parole chiave che rappresentano queste unità di misura.

Tabella 323.1. Elenco delle parole chiave che esprimono unità di misura data-orario.

Parola chiave	Significato
YEAR	Anni
MONTH	Mesi
DAY	Giorni
HOUR	Ore
MINUTE	Minuti
SECOND	Secondi

323.2.3.1 Costanti data-orario

Le costanti che rappresentano informazioni data-orario sono espresse come le stringhe, delimitate tra apici. Il sistema DBMS potrebbe ammettere più forme differenti per l'inserimento di queste, ma i modi più comuni dovrebbero essere quelli espressi dagli esempi seguenti.

```
'1999-12-31'
'12/31/1999'
'31.12.1999'
```

Questi tre esempi rappresentano la stessa data: il 31 dicembre 1999. Per una questione di uniformità, dovrebbe essere preferibile il primo di questi formati, corrispondente allo stile ISO 8601. Anche gli orari che si vedono sotto, sono aderenti allo stile ISO 8601; in particolare per il fatto che il fuso orario viene indicato attraverso lo scostamento dal tempo universale, invece che attraverso una parola chiave che definisca il fuso dell'ora locale.

```
'12:30:50+1.00'
'12:30:50.10'
'12:30:50'
'12:30'
```

Il primo di questa serie di esempi rappresenta un orario composto da ore, minuti e secondi, oltre all'indicazione dello scostamento dal tempo universale (per ottenere il tempo universale deve essere sottratta un'ora). Il secondo esempio mostra un orario composto da ore, minuti, secondi e centesimi di secondo. Il terzo e il quarto sono rappresentazioni normali, in particolare nell'ultimo è stata omessa l'indicazione dei secondi.

```
'1999-12-31 12:30:50+1.00'
'1999-12-31 12:30:50.10'
```

```
'1999-12-31 12:30:50'
'1999-12-31 12:30'
```

Gli esempi mostrano la rappresentazione di informazioni data-orario complete per il tipo '**TIMESTAMP**'. La data è separata dall'ora da uno spazio.

323.2.3.2 Costanti che esprimono intervalli

Un'informazione che rappresenta un intervallo di tempo inizia sempre con la parola chiave '**INTERVAL**' ed è seguita da una stringa che contiene l'indicazione di uno o più valori, seguiti ognuno dall'unità di misura relativi (ammesso che ciò sia necessario). Si osservino i due esempi seguenti:

```
INTERVAL '12 HOUR 30 MINUTE 50 SECOND'
INTERVAL '12:30:50'
```

Queste due forme rappresentano entrambe la stessa cosa: una durata di 12 ore, 30 minuti e 50 secondi. In generale, dovrebbe essere preferibile la seconda delle due forme di rappresentazione.

```
INTERVAL '10 DAY 12 HOUR 30 MINUTE 50 SECOND'
INTERVAL '10 DAY 12:30:50'
```

Come prima, i due esempi che si vedono sopra sono equivalenti. Intuitivamente, si può osservare che non ci può essere un altro modo di esprimere una durata in giorni, senza specificarlo attraverso la parola chiave '**DAY**'.

Per completare la serie di esempi, si aggiungono anche i casi in cui si rappresentano esplicitamente quantità molto grandi, che di conseguenza sono approssimate al mese (come richiede lo standard SQL92):

```
INTERVAL '10 YEAR 11 MONTH'
INTERVAL '10 YEAR'
```

Gli intervalli di tempo possono servire per indicare un tempo trascorso rispetto al momento attuale. Per specificare espressamente questo fatto, si indica l'intervallo come un valore negativo, aggiungendo all'inizio un trattino (il segno meno).

```
INTERVAL '- 10 YEAR 11 MONTH'
```

L'esempio che si vede sopra, esprime precisamente 10 anni e 11 mesi fa.

323.3 Operatori, funzioni ed espressioni

SQL, pur non essendo un linguaggio di programmazione completo, mette a disposizione una serie di operatori e di funzioni utili per la realizzazione di espressioni di vario tipo.

323.3.1 Operatori aritmetici

Gli operatori che intervengono su valori numerici sono elencati nella tabella 323.2.

Tabella 323.2. Elenco degli operatori aritmetici.

Operatore e operandi	Descrizione
- op	Inverte il segno dell'operando.
op1 + op2	Somma i due operandi.
op1 - op2	Sottrae dal primo il secondo operando.
*op1 * op2*	Moltiplica i due operandi.
op1 / op2	Divide il primo operando per il secondo.
op1 % op2	Modulo: il resto della divisione tra il primo e il secondo operando.

Nelle espressioni, tutti i tipi numerici esatti e approssimati possono essere usati senza limitazioni. Dove necessario, il sistema provvede a eseguire le conversioni di tipo.

323.3.2 Operazioni con i valori data-orario e con intervalli di tempo

Le operazioni che si possono compiere utilizzando valori data-orario e valori che esprimono intervalli di tempo, hanno significato solo in alcune circostanze. La tabella 323.3 elenca le operazioni possibili e il tipo di risultato che si ottiene in base al tipo di operatori utilizzato.

Tabella 323.3. Operatori e operandi validi quando si utilizzano valori data-orario e valori che esprimono intervalli di tempo.

Operatore e operandi	Risultato
data_orario - data_orario	Intervallo
data_orario +|- intervallo	Data-orario
intervallo + data_orario	Data-orario
intervallo +|- intervallo	Intervallo
*intervallo *|/ numerico*	Intervallo
*numerico * intervallo*	Intervallo

323.3.3 Operatori di confronto e operatori logici

Gli operatori di confronto determinano la relazione tra due operandi. Il risultato dell'espressione composta da due operandi posti a confronto è di tipo booleano: *Vero* o *Falso*. Gli operatori di confronto sono elencati nella tabella 323.4.

Tabella 323.4. Elenco degli operatori di confronto.

Operatore e operandi	Descrizione
op1 = op2	*Vero* se gli operandi si equivalgono.
op1 <> op2	*Vero* se gli operandi sono differenti.
op1 < op2	*Vero* se il primo operando è minore del secondo.
op1 > op2	*Vero* se il primo operando è maggiore del secondo.
op1 <= op2	*Vero* se il primo operando è minore o uguale al secondo.
op1 >= op2	*Vero* se il primo operando è maggiore o uguale al secondo.

Quando si vogliono combinare assieme diverse espressioni logiche si utilizzano gli operatori logici. Come in tutti i linguaggi di programmazione, si possono usare le parentesi tonde per raggruppare le espressioni logiche in modo da chiarire l'ordine di risoluzione. Gli operatori logici sono elencati nella tabella 323.5.

Tabella 323.5. Elenco degli operatori logici.

Operatore e operandi	Descrizione
NOT *op*	Inverte il risultato logico dell'operando.
op1 AND *op2*	*Vero* se entrambi gli operandi restituiscono il valore *Vero*.
op1 OR *op2*	*Vero* se almeno uno degli operandi restituisce il valore *Vero*.

Il meccanismo di confronto tra due operandi numerici è evidente, mentre può essere meno evidente con le stringhe di caratteri. Per la precisione, il confronto tra due stringhe avviene senza tenere conto degli spazi finali, per cui, le stringhe '`'ciao'`' e '`'ciao '`' dovrebbero risultare uguali attraverso il confronto di uguaglianza con l'operatore '`=`'.

Con le stringhe, tuttavia, si possono eseguire dei confronti basati su modelli, attraverso gli operatori '`IS LIKE`' e '`IS NOT LIKE`'. Il modello può contenere dei metacaratteri rappresentati dal trattino basso ('_'), che rappresenta un carattere qualsiasi, e dal simbolo di percentuale ('`%`'), che rappresenta una sequenza qualsiasi di caratteri. La tabella 323.6 riassume quanto affermato.

Tabella 323.6. Espressioni sulle stringhe di caratteri.

Espressioni e modelli	Descrizione
stringa IS LIKE *modello*	Restituisce *Vero* se il modello corrisponde alla stringa.
stringa IS NOT LIKE *modello*	Restituisce *Vero* se il modello non corrisponde alla stringa.
_	Rappresenta un carattere qualsiasi.
%	Rappresenta una sequenza indeterminata di caratteri.

La presenza di valori indeterminati impone la presenza di operatori di confronto in grado di determinarne l'esistenza. La tabella 323.7 riassume gli operatori ammissibili in questi casi.

Tabella 323.7. Espressioni di verifica dei valori indeterminati.

Operatori	Descrizione
espressione IS NULL	Restituisce *Vero* se l'espressione genera un risultato indeterminato.
espressione IS NOT NULL	Restituisce *Vero* se l'espressione non genera un risultato indeterminato.

Infine, occorre considerare una categoria particolare di espressioni che permettono di verificare l'appartenenza di un valore a un intervallo o a un elenco di valori. La tabella 323.8 riassume gli operatori utilizzabili.

Tabella 323.8. Espressioni per la verifica dell'appartenenza di un valore a un intervallo o a un elenco.

Operatori e operandi	Descrizione
op1 IN (*elenco*)	*Vero* se il primo operando è contenuto nell'elenco.
op1 NOT IN (*elenco*)	*Vero* se il primo operando non è contenuto nell'elenco.
op1 BETWEEN *op2* AND *op3*	*Vero* se il primo operando è compreso tra il secondo e il terzo.
op1 NOT BETWEEN *op2* AND *op3*	*Vero* se il primo operando non è compreso nell'intervallo.

323.4 Tabelle

SQL tratta le «relazioni» attraverso il modello tabellare; di conseguenza si adegua tutta la sua filosofia e il modo di esprimere i concetti nella sua documentazione. Le tabelle di SQL vengono definite nel modo seguente dalla documentazione standard.

- La tabella è un insieme di più righe. Una riga è una sequenza non vuota di valori. Ogni riga della stessa tabella ha la stessa cardinalità e contiene un valore per ogni colonna di

quella tabella. L'*i*-esimo valore di ogni riga di una tabella è un valore dell'*i*-esima colonna di quella tabella. La riga è l'elemento che costituisce la più piccola unità di dati che può essere inserita in una tabella e cancellata da una tabella.

- Il grado di una tabella è il numero di colonne della stessa. In ogni momento, il grado della tabella è lo stesso della cardinalità di ognuna delle sue righe; la cardinalità della tabella (cioè il numero delle righe contenute) è la stessa della cardinalità di ognuna delle sue colonne. Una tabella la cui cardinalità sia zero viene definita come vuota.

In pratica, la tabella è un contenitore di informazioni organizzato in righe e colonne. La tabella viene identificata per nome, così anche le colonne, mentre le righe vengono identificate attraverso il loro contenuto.

Nel modello di SQL, le colonne sono ordinate, anche se ciò non è sempre un elemento indispensabile, dal momento che si possono identificare per nome. Inoltre sono ammissibili tabelle contenenti righe duplicate.

323.4.1 Creazione di una tabella

La creazione di una tabella avviene attraverso un'istruzione che può assumere un'articolazione molto complessa, a seconda delle caratteristiche particolari che da questa tabella si vogliono ottenere. La sintassi più semplice è quella seguente:

```
CREATE TABLE nome_tabella ( specifiche )
```

Tuttavia, sono proprio le specifiche indicate tra le parentesi tonde che possono tradursi in un sistema molto confuso. La creazione di una tabella elementare può essere espressa con la sintassi seguente:

```
CREATE TABLE nome_tabella (nome_colonna tipo [,...])
```

In questo modo, all'interno delle parentesi vengono semplicemente elencati i nomi delle colonne seguiti dal tipo di dati che in esse possono essere contenuti. L'esempio seguente rappresenta l'istruzione necessaria a creare una tabella composta da cinque colonne, contenenti rispettivamente informazioni su: codice, cognome, nome, indirizzo e numero di telefono.

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40),
        Nome            char(40),
        Indirizzo       varchar(60),
        Telefono        varchar(40)
    )
```

323.4.2 Valori predefiniti

Quando si inseriscono delle righe all'interno della tabella, in linea di principio è possibile che i valori corrispondenti a colonne particolari non siano inseriti esplicitamente. Se si verifica questa situazione (purché ciò sia consentito dai vincoli), viene attribuito a questi elementi mancanti un valore predefinito. Questo può essere stabilito all'interno delle specifiche di creazione della tabella; in mancanza di tale definizione, viene attribuito '**NULL**', corrispondente al valore indefinito.

La sintassi necessaria a creare una tabella contenente le indicazioni sui valori predefiniti da utilizzare è la seguente:

```
CREATE TABLE nome_tabella (
                nome_colonna tipo
                    [DEFAULT espressione]
                [,…]
        )
```

L'esempio seguente crea la stessa tabella già vista nell'esempio precedente, specificando come valore predefinito per l'indirizzo, la stringa di caratteri: «sconosciuto».

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40),
        Nome            char(40),
        Indirizzo       varchar(60)     DEFAULT 'sconosciuto',
        Telefono        varchar(40)
    )
```

323.4.3 Vincoli interni alla tabella

Può darsi che in certe situazioni, determinati valori all'interno di una riga non siano ammissibili, a seconda del contesto a cui si riferisce la tabella. I vincoli interni alla tabella sono quelli che possono essere risolti senza conoscere informazioni esterne alla tabella stessa.

Il vincolo più semplice da esprimere è quello di non ammissibilità dei valori indefiniti. La sintassi seguente ne mostra il modo.

```
CREATE TABLE nome_tabella (
                nome_colonna tipo
                    [NOT NULL]
                [,…]
        )
```

L'esempio seguente crea la stessa tabella già vista negli esempi precedenti, specificando che il codice, il cognome, il nome e il telefono non possono essere indeterminati.

```
CREATE TABLE Indirizzi (
        Codice          integer         NOT NULL,
        Cognome         char(40)        NOT NULL,
        Nome            char(40)        NOT NULL,
        Indirizzo       varchar(60)     DEFAULT 'sconosciuto',
        Telefono        varchar(40)     NOT NULL
    )
```

Un altro vincolo importante è quello che permette di definire che un gruppo di colonne deve rappresentare dati unici in ogni riga, cioè che non siano ammissibili righe che per quel gruppo di colonne abbiano dati uguali. Segue lo schema sintattico relativo.

```
CREATE TABLE nome_tabella (
                nome_colonna tipo
                [,…],
                UNIQUE ( nome_colonna [,…] )
                [,…]
        )
```

L'indicazione dell'unicità può riguardare più gruppi di colonne in modo indipendente. Per ottenere questo si possono indicare più opzioni '**UNIQUE**'.

> È il caso di osservare che il vincolo '**UNIQUE**' non è sufficiente per impedire che i dati possano essere indeterminati. Infatti, il valore indeterminato, '**NULL**', è diverso da ogni altro '**NULL**'.

L'esempio seguente crea la stessa tabella già vista negli esempi precedenti, specificando che i dati della colonna del codice devono essere unici per ogni riga.

```
CREATE TABLE Indirizzi (
        Codice          integer         NOT NULL,
        Cognome         char(40)        NOT NULL,
        Nome            char(40)        NOT NULL,
        Indirizzo       varchar(60)     DEFAULT 'sconosciuto',
        Telefono        varchar(40)     NOT NULL,
        UNIQUE (Codice)
    )
```

Quando una colonna, o un gruppo di colonne, costituisce un riferimento importante per identificare le varie righe che compongono la tabella, si può utilizzare il vincolo '**PRIMARY KEY**', che può essere utilizzato una sola volta. Questo vincolo stabilisce anche che i dati contenuti, oltre a non poter essere doppi, non possono essere indefiniti.

```
CREATE TABLE nome_tabella (
                nome_colonna tipo
                [,...],
                PRIMARY KEY ( nome_colonna [,...] )
        )
```

L'esempio seguente crea la stessa tabella già vista negli esempi precedenti specificando che la colonna del codice deve essere considerata la chiave primaria.

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40)        NOT NULL,
        Nome            char(40)        NOT NULL,
        Indirizzo       varchar(60)     DEFAULT 'sconosciuto',
        Telefono        varchar(40)     NOT NULL,
        PRIMARY KEY (Codice)
    )
```

323.4.4 Vincoli esterni alla tabella

I vincoli esterni alla tabella riguardano principalmente la connessione con altre tabelle e la necessità che i riferimenti a queste siano validi. La definizione formale di questa connessione è molto complessa e qui non viene descritta. Si tratta, in ogni caso, dell'opzione '**FOREIGN KEY**' seguita da '**REFERENCES**'.

Vale la pena però di considerare i meccanismi che sono coinvolti. Infatti, nel momento in cui si inserisce un valore, il sistema può impedire l'operazione perché non valida in base all'assenza di quel valore in un'altra tabella esterna specificata. Il problema nasce però nel momento in cui nella tabella esterna viene eliminata o modificata una riga che era oggetto di un riferimento da parte della prima. Si pongono le alternative seguenti.

- '**CASCADE**'

  Se nella tabella esterna il dato a cui si fa riferimento è stato cambiato, viene cambiato anche il riferimento nella tabella di partenza; se nella tabella esterna la riga corrispondente viene rimossa, viene rimossa anche la riga della tabella di partenza.

- '**SET NULL**'

  Se viene a mancare l'oggetto a cui si fa riferimento, viene modificato il dato attribuendo il valore indefinito.

- '**SET DEFAULT**'

  Se viene a mancare l'oggetto a cui si fa riferimento, viene modificato il dato attribuendo il valore predefinito.

- '**NO ACTION**'

  Se viene a mancare l'oggetto a cui si fa riferimento, non viene modificato il dato contenuto nella tabella di partenza.

Le azioni da compiere si possono distinguere in base all'evento che ha causato la rottura del riferimento: cancellazione della riga della tabella esterna o modifica del suo contenuto.

323.4.5 Modifica della struttura della tabella

La modifica della struttura di una tabella riguarda principalmente la sua organizzazione in colonne. Le cose più semplici che si possono desiderare di fare sono l'aggiunta di nuove colonne e l'eliminazione di colonne esistenti. Vedendo il problema in questa ottica, la sintassi si riduce ai due casi seguenti.

```
ALTER TABLE nome_tabella (
            ADD [COLUMN] nome_colonna tipo [altre_caratteristiche]
        )

ALTER TABLE nome_tabella (
                DROP [COLUMN] nome_colonna
        )
```

Nel primo caso si aggiunge una colonna, della quale si deve specificare il nome, il tipo ed eventualmente i vincoli; nel secondo si tratta solo di indicare la colonna da eliminare. A livello di singola colonna può essere eliminato o attribuito un valore predefinito.

```
ALTER TABLE nome_tabella (
            ALTER [COLUMN] nome_colonna DROP DEFAULT
        )

ALTER TABLE nome_tabella (
                ALTER [COLUMN] nome_colonna SET DEFAULT valore_predefinito
        )
```

323.4.6 Eliminazione di una tabella

L'eliminazione di una tabella, con tutto il suo contenuto, è un'operazione semplice che dovrebbe essere autorizzata solo all'utente che l'ha creata.

```
DROP TABLE nome_tabella
```

323.5 Inserimento, eliminazione e modifica dei dati

L'inserimento, l'eliminazione e la modifica dei dati di una tabella è un'operazione che interviene sempre a livello delle righe. Infatti, come già definito, la riga è l'elemento che costituisce l'unità di dati più piccola che può essere inserita o cancellata da una tabella.

323.5.1 Inserimento di righe

L'inserimento di una nuova riga all'interno di una tabella viene eseguito attraverso l'istruzione '**INSERT**'. Dal momento che nel modello di SQL le colonne sono ordinate, è sufficiente indicare ordinatamente l'elenco dei valori della riga da inserire, come mostra la sintassi seguente:

```
INSERT INTO nome_tabella VALUES (espressione_1 [,…espressione_N])
```

Per esempio, l'inserimento di una riga nella tabella '**Indirizzi**' già mostrata in precedenza, potrebbe avvenire nel modo seguente:

```
INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )
```

Se i valori inseriti sono meno del numero delle colonne della tabella, i valori mancanti, in coda, ottengono quanto stabilito come valore predefinito, o '**NULL**' in sua mancanza (sempre che ciò sia concesso dai vincoli della tabella).

L'inserimento dei dati può avvenire in modo più chiaro e sicuro elencando prima i nomi delle colonne, in modo da evitare di dipendere dalla sequenza delle colonne memorizzata nella tabella. La sintassi seguente mostra il modo di ottenere questo.

```
INSERT INTO nome_tabella [(colonna_1 [,...colonna_N])]
    VALUES (espressione_1 [,...espressione_N]);
```

L'esempio già visto potrebbe essere tradotto nel modo seguente, più prolisso, ma anche più chiaro.

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
        Indirizzo,
        Telefono
    )
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )
```

Questo modo esplicito di fare riferimento alle colonne garantisce anche che eventuali modifiche di lieve entità nella struttura della tabella non debbano necessariamente riflettersi nei programmi. L'esempio seguente mostra l'inserimento di alcuni degli elementi della riga, lasciando che gli altri ottengano l'assegnamento di un valore predefinito.

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
        Telefono
    )
    VALUES (
        01,
        'Pinco',
        'Pallino',
        '0222,222222'
    )
```

323.5.2 Aggiornamento delle righe

La modifica delle righe può avvenire attraverso una scansione della tabella, dalla prima all'ultima riga, eventualmente controllando la modifica in base all'avverarsi di determinate condizioni. La sintassi per ottenere questo risultato, leggermente semplificata, è la seguente:

```
UPDATE tabella
    SET colonna_1=espressione_1 [ ,…colonna_N=espressione_N ]
    [WHERE condizione]
```

L'istruzione '**UPDATE**' esegue tutte le sostituzioni indicate dalle coppie *colonna=espressione*, per tutte le righe in cui la condizione posta dopo la parola chiave '**WHERE**' si avvera. Se tale condizione manca, l'effetto delle modifiche si riflette su tutte le righe della tabella.

L'esempio seguente aggiunge una colonna alla tabella degli indirizzi, per contenere il nome del comune di residenza; successivamente viene inserito il nome del comune «Sferopoli» in base al prefisso telefonico.

```
ALTER TABLE Indirizzi ADD COLUMN Comune char(30)

UPDATE Indirizzi
    SET Comune='Sferopoli'
    WHERE Telefono >= '022' AND Telefono < '023'
```

Eventualmente, al posto dell'espressione si può indicare la parola chiave '**DEFAULT**' che fa in modo di assegnare il valore predefinito per quella colonna.

323.5.3 Eliminazione di righe

La cancellazione di righe da una tabella è un'operazione molto semplice. Richiede solo l'indicazione del nome della tabella e la condizione in base alla quale le righe devono essere cancellate.

```
DELETE FROM tabella [WHERE condizione]
```

> Se la condizione non viene indicata, si cancellano tutte le righe!

323.6 Interrogazioni di tabelle

L'interrogazione di una tabella è l'operazione con cui si ottengono i dati contenuti al suo interno, in base a dei criteri di filtro determinati. L'interrogazione consente anche di combinare assieme dati provenienti da tabelle differenti, in base a delle relazioni che possono intercorrere tra queste.

323.6.1 Interrogazioni elementari

La forma più semplice di esprimere la sintassi necessaria a interrogare **una** sola tabella è quella espressa dallo schema seguente:

```
SELECT espress_col_1 [ ,…espress_col_N ]
    FROM tabella
    [WHERE condizione]
```

In questo modo è possibile definire le colonne che si intendono utilizzare per il risultato, mentre le righe si specificano, eventualmente, con la condizione posta dopo la parola chiave '**WHERE**'.

L'esempio seguente mostra la proiezione delle colonne del cognome e nome della tabella di indirizzi già vista negli esempi delle altre sezioni, senza porre limiti alle righe.

```
SELECT Cognome, Nome FROM Indirizzi
```

Quando si vuole ottenere una selezione composta dalle stesse colonne della tabella originale, nel suo stesso ordine, si può utilizzare un carattere jolly particolare, l'asterisco ('**'). Questo rappresenta l'elenco di tutte le colonne della tabella indicata.

```
SELECT * FROM Indirizzi
```

È bene osservare che le colonne si esprimono attraverso un'espressione, questo significa che le colonne a cui si fa riferimento sono quelle del risultato finale, cioè della tabella che viene restituita come selezione o proiezione della tabella originale. L'esempio seguente emette una sola colonna contenente un ipotetico prezzo scontato del 10 %, in pratica viene moltiplicato il valore di una colonna contenente il prezzo per 0,90, in modo da ottenerne il 90 % (100 % meno lo sconto).

```
SELECT Prezzo * 0.90 FROM Listino
```

In questo senso si può comprendere l'utilità di attribuire esplicitamente un nome alle colonne del risultato finale, come indicato dalla sintassi seguente:

```
SELECT espress_col_1 AS nome_col_1 [,…espress_col_N AS nome_col_N]
    FROM tabella
    [WHERE condizione]
```

In questo modo, l'esempio precedente può essere trasformato come segue, dando un nome alla colonna generata e chiarendone così il contenuto.

```
SELECT Prezzo * 0.90 AS Prezzo_Scontato FROM Listino
```

Finora è stata volutamente ignorata la condizione che controlla le righe da selezionare. Anche se potrebbe essere evidente, è bene chiarire che la condizione posta dopo la parola chiave '**WHERE**' può fare riferimento solo ai dati originali della tabella da cui si attingono. Quindi, non è valida una condizione che utilizza un riferimento a un nome che appare dopo la parola chiave '**AS**' abbinata alle espressioni delle colonne.

Per qualche motivo che verrà chiarito in seguito, può essere conveniente attribuire un alias alla tabella da cui estrarre i dati. Anche in questo caso si utilizza la parola chiave '**AS**', come indicato dalla sintassi seguente:

```
SELECT specificazione_della_colonna_1 [,…specificazione_della_colonna_N]
    FROM tabella AS alias
    [WHERE condizione]
```

Quando si vuole fare riferimento al nome di una colonna, se per qualche motivo questo nome dovesse risultare ambiguo, si può aggiungere anteriormente il nome della tabella a cui appartiene, separandolo attraverso l'operatore punto ('**.**'). L'esempio seguente è la proiezione dei cognomi e dei nomi della solita tabella degli indirizzi. In questo caso, le espressioni delle colonne rappresentano solo le colonne corrispondenti della tabella originaria, con l'aggiunta dell'indicazione esplicita del nome della tabella stessa.

```
SELECT Indirizzi.Cognome, Indirizzi.Nome FROM Indirizzi
```

A questo punto, se al nome della tabella viene abbinato un alias, si può esprimere la stessa cosa indicando il nome dell'alias al posto di quello della tabella, come nell'esempio seguente:

```
SELECT Ind.Cognome, Ind.Nome FROM Indirizzi AS Ind
```

323.6.2 Interrogazioni ordinate

Per ottenere un elenco ordinato in base a qualche criterio, si utilizza l'istruzione '**SELECT**' con l'indicazione di un'espressione in base alla quale effettuare l'ordinamento. Questa espressione è preceduta dalle parole chiave '**ORDER BY**':

```
SELECT espress_col_1 [ ,…espress_col_N ]
    FROM tabella
    [ WHERE condizione ]
    ORDER BY espressione [ ASC | DESC ] [ ,… ]
```

L'espressione può essere il nome di una colonna, oppure un'espressione che genera un risultato da una o più colonne; l'aggiunta eventuale della parola chiave '**ASC**', o '**DESC**', permette di specificare un ordinamento crescente, o discendente. Come si vede, le espressioni di ordinamento possono essere più di una, separate con una virgola.

```
SELECT Cognome, Nome FROM Indirizzi ORDER BY Cognome
```

L'esempio mostra un'applicazione molto semplice del problema, in cui si ottiene un elenco delle sole colonne '**Cognome**' e '**Nome**', della tabella '**Indirizzi**', ordinato per '**Cognome**'.

```
SELECT Cognome, Nome FROM Indirizzi ORDER BY Cognome, Nome
```

Questo esempio, aggiunge l'indicazione del nome nella chiave di ordinamento, in modo che in presenza di cognomi uguali, la scelta venga fatta in base al nome.

```
SELECT Cognome, Nome FROM Indirizzi ORDER BY TRIM( Cognome ), TRIM( Nome )
```

Questo ultimo esempio mostra l'utilizzo di due espressioni come chiave di ordinamento. Per la precisione, la funzione '**TRIM()**', usata in questo modo, serve a eliminare gli spazi iniziali e finali superflui. In questo modo, se i nomi e i cognomi sono stati inseriti con degli spazi iniziali, questi non vanno a influire sull'ordinamento.

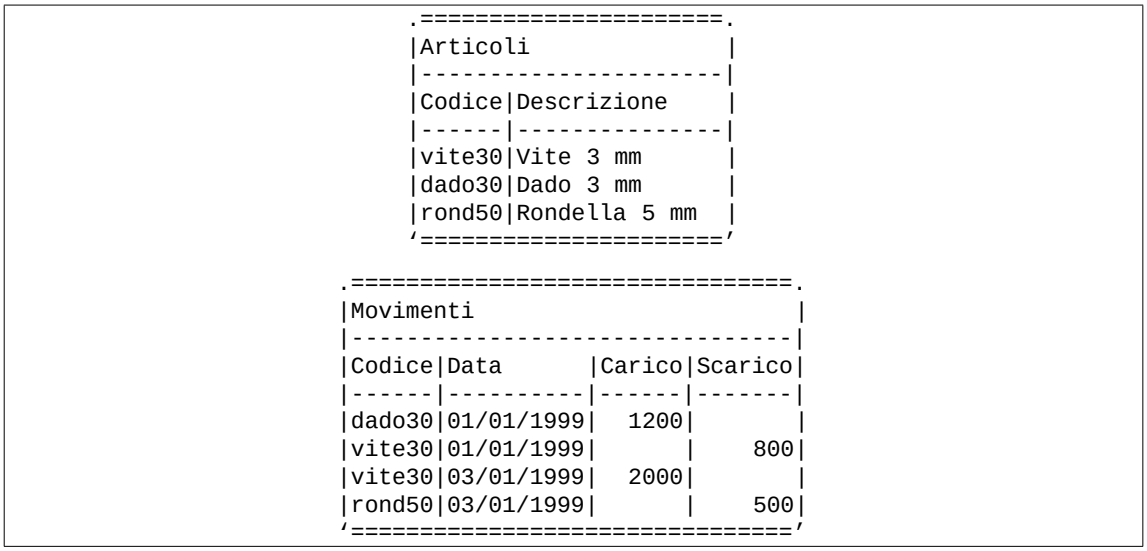
323.6.3 Interrogazioni simultanee di più tabelle

Se dopo la parola chiave '**FROM**' si indicano più tabelle (ciò vale anche se si indica più volte la stessa tabella), si intende fare riferimento a una tabella generata dal prodotto di queste. Se per esempio si vogliono abbinare due tabelle, una di tre righe per due colonne e un'altra di due righe per due colonne, quello che si ottiene sarà una tabella di quattro colonne composta da sei righe. Infatti, ogni riga della prima tabella risulta abbinata con ogni riga della seconda.

```
SELECT specificazione_della_colonna_1 [ ,…specificazione_della_colonna_N ]
    FROM specificazione_della_tabella_1 [ ,…specificazione_della_tabella_N ]
    [ WHERE condizione ]
```

Nel capitolo precedente è stato mostrato un esempio di gestione del magazzino. Vengono riproposte le tabelle di quell'esempio, ancora più semplificate (figura 323.1).

Figura 323.1. Tabelle '**Articoli**' e '**Movimenti**' di una gestione del magazzino ipotetica.



Da questa situazione si vuole ottenere il join della tabella '**Movimenti**' con tutte le informazioni corrispondenti della tabella '**Articoli**', basando il riferimento sulla colonna '**Codice**'. In pratica si vuole ottenere la tabella della figura 323.2.

Tabella 323.8. Risultato del join che si intende ottenere tra la tabella '**Movimenti**' e la tabella '**Articoli**'.

Codice	Data	Carico	Scarico	Descrizione
dado30	01/01/1999	1200		Dado 3 mm
vite30	01/01/1999		800	Vite 3 mm
vite30	03/01/1999	2000		Vite 3 mm
rond50	03/01/1999		500	Rondella 5 mm

Considerato che da un'istruzione '**SELECT**' contenente il riferimento a più tabelle si genera il prodotto tra queste, si pone poi il problema di eseguire una proiezione delle colonne desiderate e, soprattutto, di selezionare le righe. In questo caso, la selezione deve essere basata sulla corrispondenza tra la colonna '**Codice**' della prima tabella, con la stessa colonna della seconda. Dovendo fare riferimento a due colonne di tabelle differenti, aventi però lo stesso nome, diviene indispensabile indicare i nomi delle colonne prefissandoli con i nomi delle tabelle rispettive.

```
SELECT
    Movimenti.Codice,
    Movimenti.Data,
    Movimenti.Carico,
    Movimenti.Scarico,
    Articoli.Descrizione
    FROM Movimenti, Articoli
    WHERE Movimenti.Codice = Articoli.Codice;
```

L'interrogazione simultanea di più tabelle si presta anche per elaborazioni della stessa tabella più volte. In tal caso, diventa obbligatorio l'uso degli alias. Si osservi il caso seguente:

```
SELECT Ind1.Cognome, Ind1.Nome
    FROM Indirizzi AS Ind1, Indirizzi AS Ind2
    WHERE
        Ind1.Cognome = Ind2.Cognome
    AND Ind1.Nome <> Ind2.Nome
```

Il senso di questa interrogazione, che utilizza la stessa tabella degli indirizzi per due volte con due alias differenti, è quello di ottenere l'elenco delle persone che hanno lo stesso cognome, avendo

però un nome differente.

Esiste anche un'altra situazione in cui si ottiene l'interrogazione simultanea di più tabelle: l'*unione* . Si tratta semplicemente di attaccare il risultato di un'interrogazione su una tabella con quello di un'altra tabella, quando le colonne finali appartengono allo stesso tipo di dati.

```
SELECT specificazione_della_colonna_1 [ , ...specificazione_della_colonna_N ]
        FROM specificazione_della_tabella_1 [ , ...specificazione_della_tabella_N ]
        [ WHERE condizione ]
    UNION
        SELECT specificazione_della_colonna_1 [ , ...specificazione_della_colonna_N ]
            FROM specificazione_della_tabella_1 [ , ...specificazione_della_tabella_N ]
            [ WHERE condizione ]
```

Lo schema sintattico dovrebbe essere abbastanza esplicito: si uniscono due istruzioni '**SELECT**' in un risultato unico, attraverso la parola chiave '**UNION**'.

323.6.4 Condizioni

La condizione che esprime la selezione delle righe può essere composta come si vuole, purché il risultato sia di tipo logico e i dati a cui si fa riferimento provengano dalle tabelle di partenza. Quindi si possono usare anche altri operatori di confronto, funzioni e operatori booleani.

> È bene ricordare che il valore indefinito, rappresentato da '**NULL**', è diverso da qualunque altro valore, compreso un altro valore indefinito. Per verificare che un valore sia o non sia indefinito, si deve usare l'operatore '**IS NULL**' oppure '**IS NOT NULL**'.

323.6.5 Aggregazioni

L'aggregazione è una forma di interrogazione attraverso cui si ottengono risultati riepilogativi del contenuto di una tabella, in forma di tabella contenente una sola riga. Per questo si utilizzano delle funzioni speciali al posto dell'espressione che esprime le colonne del risultato. Queste funzioni restituiscono un solo valore e come tali concorrono a creare un'unica riga. Le funzioni di aggregazione sono: '**COUNT()**', '**SUM()**', '**MAX()**', '**MIN()**', '**AVG()**'. Per intendere il problema, si osservi l'esempio seguente:

```
SELECT COUNT(*) FROM Movimenti WHERE <synellipsis>
```

In questo caso, quello che si ottiene è solo il numero di righe della tabella '**Movimenti**' che soddisfano la condizione posta dopo la parola chiave '**WHERE**' (qui non è stata indicata). L'asterisco posto come parametro della funzione '**COUNT()**' rappresenta effettivamente l'elenco di tutti i nomi delle colonne della tabella '**Movimenti**'.

Quando si utilizzano funzioni di questo tipo, occorre considerare che l'elaborazione si riferisce alla tabella virtuale generata dopo la selezione posta da '**WHERE**'.

La funzione '**COUNT()**' può essere descritta attraverso la sintassi seguente:

```
COUNT( * )
```

```
COUNT( [ DISTINCT | ALL ] lista_colonne )
```

Utilizzando la forma già vista, quella dell'asterisco, si ottiene solo il numero delle righe della tabella. L'opzione '**DISTINCT**', seguita da una lista di nomi di colonne, fa in modo che vengano contate le righe contenenti valori differenti per quel gruppo di colonne. L'opzione '**ALL**' è implicita quando non si usa '**DISTINCT**' e indica semplicemente di contare tutte le righe.

Il conteggio delle righe esclude in ogni caso quelle in cui il contenuto di tutte le colonne selezionate è indefinito ('**NULL**').

Le altre funzioni aggreganti non prevedono l'asterisco, perché fanno riferimento a un'espressione che genera un risultato per ogni riga ottenuta dalla selezione.

```
SUM( [DISTINCT|ALL] espressione )
```

```
MAX( [DISTINCT|ALL] espressione )
```

```
MIN( [DISTINCT|ALL] espressione )
```

```
AVG( [DISTINCT|ALL] espressione )
```

In linea di massima, per tutti questi tipi di funzioni aggreganti, l'espressione deve generare un risultato numerico, sul quale calcolare la sommatoria, '**SUM()**', il valore massimo, '**MAX()**', il valore minimo, '**MIN()**', la media, '**AVG()**'.

L'esempio seguente calcola lo stipendio medio degli impiegati, ottenendo i dati da un'ipotetica tabella '**Emolumenti**', limitandosi ad analizzare le righe riferite a un certo settore.

```
SELECT AVG( Stipendio ) FROM Emolumenti
    WHERE Settore = 'Amministrazione'
```

L'esempio seguente è una variante in cui si estraggono rispettivamente lo stipendio massimo, medio e minimo.

```
SELECT MAX( Stipendio ), AVG( Stipendio ), MIN( Stipendio ) FROM Emolumenti
    WHERE Settore = 'Amministrazione'
```

L'esempio seguente è invece volutamente **errato**, perché si mescolano funzioni aggreganti assieme a espressioni di colonna normali.

```
-- Esempio errato
SELECT MAX( Stipendio ), Settore FROM Emolumenti
    WHERE Settore = 'Amministrazione'
```

323.6.6 Raggruppamenti

Le aggregazioni possono essere effettuate in riferimento a gruppi di righe, distinguibili in base al contenuto di una o più colonne. In questo tipo di interrogazione si può generare solo una tabella composta da tante colonne quante sono quelle prese in considerazione dalla clausola di raggruppamento, assieme ad altre contenenti solo espressioni di aggregazione.

Alla sintassi normale già vista nelle sezioni precedenti, si aggiunge la clausola '**GROUP BY**'.

```
SELECT specificazione_della_colonna_1 [,…specificazione_della_colonna_N ]
    FROM specificazione_della_tabella_1 [,…specificazione_della_tabella_N ]
    [WHERE condizione ]
    GROUP BY colonna_1 [,… ]
```

Per comprendere l'effetto di questa sintassi, si deve scomporre idealmente l'operazione di selezione da quella di raggruppamento:

1. la tabella ottenuta dall'istruzione '**SELECT**…**FROM**' viene filtrata dalla condizione '**WHERE**';
2. la tabella risultante viene riordinata in modo da raggruppare le righe in cui i contenuti delle colonne elencate dopo la clausola '**GROUP BY**' sono uguali;
3. su questi gruppi di righe vengono valutate le funzioni di aggregazione.

Figura 323.3. Carichi e scarichi in magazzino.

Movimenti				
Codice	Data	Carico	Scarico	...
vite40	01/01/1999	1200		...
vite30	01/01/1999		800	...
vite40	01/01/1999	1500		...
vite30	02/01/1999		1000	...
vite30	03/01/1999	2000		...
rond50	03/01/1999		500	...
vite40	04/01/1999	2200		...

Si osservi la tabella riportata in figura 323.3, mostra la solita sequenza di carichi e scarichi di magazzino. Si potrebbe porre il problema di conoscere il totale dei carichi e degli scarichi per ogni articolo di magazzino. La richiesta può essere espressa con l'istruzione seguente:

```
SELECT Codice, SUM( Carico ), SUM( Scarico ) FROM Movimenti
    GROUP BY Codice
```

Quello che si ottiene appare nella figura 323.4.

Figura 323.4. Carichi e scarichi in magazzino.

Codice	SUM(Carico)	SUM(Scarico)
vite40	4900	
vite30	2000	1800
rond50		500

Volendo si possono fare i raggruppamenti in modo da avere i totali distinti anche in base al giorno, come nell'istruzione seguente:

```
SELECT Codice, Data, SUM( Carico ), SUM( Scarico ) FROM Movimenti
    GROUP BY Codice, Data
```

Come già affermato, la condizione posta dopo la parola chiave '**`WHERE`**' serve a filtrare inizialmente le righe da considerare nel raggruppamento. Se quello che si vuole è filtrare ulteriormente il risultato di un raggruppamento, occorre usare la clausola '**`HAVING`**'.

```
SELECT specificazione_della_colonna_1 [,...specificazione_della_colonna_N ]
    FROM specificazione_della_tabella_1 [,...specificazione_della_tabella_N ]
    [WHERE condizione ]
    GROUP BY colonna_1 [,...]
    HAVING condizione
```

L'esempio seguente serve a ottenere il raggruppamento dei carichi e scarichi degli articoli, limitando però il risultato a quelli per i quali sia stata fatta una quantità di scarichi consistente (superiore a 1000 unità).

```
SELECT Codice, SUM( Carico ), SUM( Scarico ) FROM Movimenti
    GROUP BY Codice
    HAVING SUM( Scarico ) > 1000
```

Dall'esempio già visto in figura 323.4 risulterebbe escluso l'articolo '**`rond50`**'.

323.7 Trasferimento di dati in un'altra tabella

Alcune forme particolari di richieste SQL possono essere utilizzate per inserire dati in tabelle esistenti o per crearne di nuove.

323.7.1 Creazione di una nuova tabella a partire da altre

L'istruzione '**SELECT**' può servire per creare una nuova tabella a partire dai dati ottenuti dalla sua interrogazione.

```
SELECT specificazione_della_colonna_1 [,...specificazione_della_colonna_N ]
    INTO TABLE tabella_da_generare
    FROM specificazione_della_tabella_1 [,...specificazione_della_tabella_N ]
    [WHERE condizione ]
```

L'esempio seguente crea la tabella '**Mia_prova**' come risultato della fusione delle tabelle '**Indirizzi**' e '**Presenze**'.

```
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    INTO TABLE Mia_prova
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

323.7.2 Inserimento in una tabella esistente

L'inserimento di dati in una tabella esistente prelevando da dati contenuti in altre, può essere fatta attraverso l'istruzione '**INSERT**' sostituendo la clausola '**VALUES**' con un'interrogazione ('**SELECT**').

```
INSERT INTO nome_tabella [(colonna_1 ...colonna_N )]
    SELECT espressione_1, ... espressione_N
        FROM tabelle_di_origine
        [WHERE condizione ]
```

L'esempio seguente aggiunge alla tabella dello storico delle presenze le registrazioni vecchie che poi vengono cancellate.

```
INSERT INTO PresenzeStorico (
        PresenzeStorico.Codice,
        PresenzeStorico.Giorno,
        PresenzeStorico.Ingresso,
        PresenzeStorico.Uscita
    )
    SELECT
        Presenze.Codice,
        Presenze.Giorno,
        Presenze.Ingresso,
        Presenze.Uscita
        FROM Presenze
        WHERE Presenze.Giorno <= '01/01/1999';

DELETE FROM Presenze WHERE Giorno <= '01/01/1999';
```

323.8 Viste

Le viste sono delle tabelle virtuali ottenute a partire da tabelle vere e proprie o da altre viste, purché non si formino ricorsioni. Il concetto non dovrebbe risultare strano. In effetti, il risultato delle interrogazioni è sempre in forma di tabella. La vista crea una sorta di interrogazione permanente che acquista la personalità di una tabella normale.

```
CREATE VIEW nome_vista [(colonna_1[,...colonna_N)]]
    AS richiesta
```

Dopo la parola chiave '**AS**' deve essere indicato ciò che compone un'istruzione '**SELECT**'. L'esempio seguente, genera la vista dei movimenti di magazzino del solo articolo '**vite30**'.

```
CREATE VIEW Movimenti_Vite30
    AS SELECT Codice, Data, Carico, Scarico
        FROM Movimenti
        WHERE Codice = 'vite30'
```

L'eliminazione di una vista si ottiene con l'istruzione '**DROP VIEW**', come illustrato dallo schema sintattico seguente:

```
DROP VIEW nome_vista
```

Volendo eliminare la vista '**Movimenti_Vite30**', si può intervenire semplicemente come nell'esempio seguente:

```
DROP VIEW Movimenti_Vite30
```

323.9 Controllare gli accessi

La gestione degli accessi in una base di dati è molto importante e potenzialmente indipendente dall'eventuale gestione degli utenti del sistema operativo sottostante. Per quanto riguarda i sistemi Unix, il DBMS può riutilizzare la definizione degli utenti del sistema operativo, farvi riferimento, oppure astrarsi completamente.

Un DBMS SQL richiede la presenza di un DBA (*Data base administrator*) che in qualità di amministratore ha sempre tutti i privilegi necessari a intervenire come vuole nel DBMS. Il nome simbolico predefinito per questo utente dal linguaggio SQL è '**_SYSTEM**'.

Il sistema di definizione degli utenti è esterno al linguaggio SQL, perché SQL si occupa solo di stabilire i privilegi legati alle tabelle.

323.9.1 Creatore

L'utente che crea una tabella, o un'altra risorsa, è il suo creatore. Su tale risorsa è l'unico utente che possa modificarne la struttura e che possa eliminarla. In pratica è l'unico che possa usare le istruzioni '**DROP**' e '**ALTER**'. Chi crea una tabella, o un'altra risorsa, può concedere o revocare i privilegi degli altri utenti su di essa.

323.9.2 Tipi di privilegi

I privilegi che si possono concedere o revocare su una risorsa sono di vario tipo, espressi attraverso una parola chiave particolare. È bene considerare i casi seguenti:

- '**SELECT**' -- rappresenta l'operazione di lettura del valore di un oggetto della risorsa, per esempio dei valori di una riga da una tabella (in pratica si riferisce all'uso dell'istruzione '**SELECT**');

- '**INSERT**' -- rappresenta l'azione di inserire un nuovo oggetto nella risorsa, come l'inserimento di una riga in una tabella;
- '**UPDATE**' -- rappresenta l'operazione di aggiornamento del valore di un oggetto della risorsa, per esempio la modifica del contenuto di una riga di una tabella;
- '**DELETE**' -- rappresenta l'eliminazione di un oggetto dalla risorsa, come la cancellazione di una riga da una tabella;
- '**ALL PRIVILEGES**' -- rappresenta simultaneamente tutti i privilegi possibili riferiti a un oggetto.

323.9.3 Concedere i privilegi

I privilegi su una tabella, o su un'altra risorsa, vengono concessi attraverso l'istruzione '**GRANT**'.

```
GRANT privilegi
    ON risorsa [,…]
    TO utenti
    [WITH GRANT OPTION]
```

Nella maggior parte dei casi, le risorse da controllare coincidono con una tabella. L'esempio seguente permette all'utente '**Pippo**' di leggere il contenuto della tabella '**Movimenti**'.

```
GRANT SELECT ON Movimenti TO Pippo
```

L'esempio seguente, concede tutti i privilegi sulla tabella '**Movimenti**' agli utenti '**Pippo**' e '**Arturo**'.

```
GRANT ALL PRIVILEGES ON Movimenti TO Pippo, Arturo
```

L'opzione '**WITH GRANT OPTION**' permette agli utenti presi in considerazione di concedere a loro volta tali privilegi ad altri utenti. L'esempio seguente concede all'utente '**Pippo**' di accedere in lettura al contenuto della tabella '**Movimenti**' e gli permette di concedere lo stesso privilegio ad altri.

```
GRANT SELECT ON Movimenti TO Pippo WITH GRANT OPTION
```

323.9.4 Revocare i privilegi

I privilegi su una tabella, o un'altra risorsa, vengono revocati attraverso l'istruzione '**REVOKE**'.

```
REVOKE privilegi
    ON risorsa [,…]
    FROM utenti
```

L'esempio seguente toglie all'utente '**Pippo**' il permesso di accedere in lettura al contenuto della tabella '**Movimenti**'.

```
REVOKE SELECT ON Movimenti FROM Pippo
```

L'esempio seguente toglie tutti i privilegi sulla tabella '**Movimenti**' agli utenti '**Pippo**' e '**Arturo**'.

```
REVOKE ALL PRIVILEGES ON Movimenti FROM Pippo, Arturo
```

323.10 Controllo delle transazioni

Una transazione SQL, è una sequenza di istruzioni che rappresenta un corpo unico dal punto di vista della memorizzazione effettiva dei dati. In altre parole, secondo l'SQL, la registrazione delle modifiche apportate alla base di dati avviene in modo asincrono, raggruppando assieme l'effetto di gruppi di istruzioni determinati.

Una transazione inizia nel momento in cui l'interprete SQL incontra delle istruzioni determinate, terminando con l'istruzione '**COMMIT**', oppure '**ROLLBACK**': nel primo caso si conferma la transazione che viene memorizzata regolarmente, mentre nel secondo si richiede di annullare le modifiche apportate dalla transazione:

```
COMMIT [WORK]
```

```
ROLLBACK [WORK]
```

Stando così le cose, si intende la necessità di utilizzare regolarmente l'istruzione '**COMMIT**' per memorizzare i dati quando non esiste più la necessità di annullare le modifiche.

```
COMMIT

INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )

COMMIT
```

L'esempio mostra un uso intensivo dell'istruzione '**COMMIT**', dove dopo l'inserimento di una riga nella tabella '**Indirizzi**', viene confermata immediatamente la transazione.

```
COMMIT

INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )

ROLLBACK
```

Questo esempio mostra un ripensamento (per qualche motivo). Dopo l'inserimento di una riga nella tabella '**Indirizzi**', viene annullata la transazione, riportando la tabella allo stato precedente.

323.11 Cursori

Quando il risultato di un'interrogazione SQL deve essere gestito all'interno di un programma, si pone un problema nel momento in cui ciò che si ottiene è più di una sola riga. Per poter scorrere un elenco ottenuto attraverso un'istruzione '**SELECT**', riga per riga, si deve usare un *cursore*.

La dichiarazione e l'utilizzo di un cursore avviene all'interno di una transazione. Quando la transazione si chiude attraverso un '**COMMIT**' o un '**ROLLBACK**', si chiude anche il cursore.

323.11.1 Dichiarazione e apertura

L'SQL prevede due fasi prima dell'utilizzo di un cursore: la dichiarazione e la sua apertura:

```
DECLARE cursore [INSENSITIVE] [SCROLL] CURSOR FOR
    SELECT …
```

```
OPEN cursore
```

Nella dichiarazione, la parola chiave '**INSENSITIVE**' serve a stabilire che il risultato dell'interrogazione che si scandisce attraverso il cursore, non deve essere sensibile alle variazioni dei dati originali; la parola chiave '**SCROLL**' indica che è possibile estrarre più righe simultaneamente attraverso il cursore.

```
DECLARE Mio_cursore CURSOR FOR
    SELECT
        Presenze.Giorno,
        Presenze.Ingresso,
        Presenze.Uscita,
        Indirizzi.Cognome,
        Indirizzi.Nome
        FROM Presenze, Indirizzi
        WHERE Presenze.Codice = Indirizzi.Codice;
```

L'esempio mostra la dichiarazione del cursore '**Mio_cursore**', abbinato alla selezione delle colonne composte dal collegamento di due tabelle, '**Presenze**' e '**Indirizzi**', dove le righe devono avere lo stesso numero di codice. Per attivare questo cursore, lo si deve aprire come nell'esempio seguente:

```
OPEN Mio_cursore
```

323.11.2 Scansione

La scansione di un'interrogazione inserita in un cursore, avviene attraverso l'istruzione '**FETCH**'. Il suo scopo è quello di estrarre una riga alla volta, in base a una posizione, relativa o assoluta.

```
FETCH [ [ NEXT | PRIOR | FIRST | LAST | { ABSOLUTE | RELATIVE } n ]
    FROM cursore ]
    INTO :variabile [,…]
```

Le parole chiave '**NEXT**', '**PRIOR**', '**FIRST**', '**LAST**', permettono rispettivamente di ottenere la riga successiva, quella precedente, la prima e l'ultima. Le parole chiave '**ABSOLUTE**' e '**RELATIVE**' sono seguite da un numero, corrispondente alla scelta della riga *n*-esima, rispetto all'inizio del gruppo per il quale è stato definito il cursore ('**ABSOLUTE**'), oppure della riga *n*-esima rispetto all'ultima riga estratta da un'istruzione '**FETCH**' precedente.

Le variabili indicate dopo la parola chiave '**INTO**', che in particolare sono precedute da due punti ('**:**'), ricevono ordinatamente il contenuto delle varie colonne della riga estratta. Naturalmente, le variabili in questione devono appartenere a un linguaggio di programmazione che incorpora l'SQL, dal momento che l'SQL stesso non fornisce questa possibilità.

```
FETCH NEXT FROM Mio_cursore
```

L'esempio mostra l'uso tipico di questa istruzione, dove si legge la riga successiva (se non ne sono state lette fino a questo punto, si tratta della prima), dal cursore dichiarato e aperto precedentemente. L'esempio seguente è identico dal punto di vista funzionale.

```
FETCH RELATIVE 1 FROM Mio_cursore
```

I due esempi successivi sono equivalenti e servono a ottenere la riga precedente.

```
FETCH PRIOR FROM Mio_cursore
```

```
FETCH RELATIVE -1 FROM Mio_cursore
```

323.11.3 Chiusura

Il cursore, al termine dell'utilizzo, deve essere chiuso:

```
CLOSE cursore
```

Seguendo gli esempi visti in precedenza, per chiudere il cursore '**`Mio_cursore`**' basta l'istruzione seguente:

```
CLOSE Mio_cursore
```

323.12 Riferimenti

- Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Riccardo Torlone, *Basi di dati, concetti, linguaggi e architetture*, McGraw-Hill
- James Hoffmann, *Introduction to Structured Query Language*

  <http://www.highcroft.com/highcroft/sql_intro.pdf>
- *SQL Standard Home Page*

  <http://www.jcc.com/sql_stnd.html>
- *SQL Reference Page*

  <http://www.contrib.andrew.cmu.edu/~shadow/sql.html>
- *ISO/IEC 9075:1992, Database Language SQL*

  <http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt>

PostgreSQL: struttura e preparazione

PostgreSQL [1] è un DBMS (*Data base management system*) relazionale esteso agli oggetti. In questo capitolo si vuole introdurre al suo utilizzo e accennare alla sua struttura, senza affrontare le particolarità del linguaggio di interrogazione. Il nome lascia intendere che si tratti di un DBMS in grado di comprendere le istruzioni SQL, anche se per il momento l'aderenza a quello standard è solo parziale.

324.1 Struttura dei dati nel file system

PostgreSQL, a parte i programmi binari, gli script e la documentazione, colloca i file di gestione delle basi di dati a partire da una certa directory, che nella documentazione originale viene definita '**`PGDATA`**'. Questo è il nome di una variabile di ambiente che può essere utilizzato per informare i vari programmi di PostgreSQL della sua collocazione; tuttavia, di solito questo meccanismo della variabile di ambiente non viene utilizzato, specificando tale directory in fase di compilazione dei sorgenti.

Questa directory corrisponde solitamente anche alla directory iniziale dell'utente di sistema per l'amministrazione di PostgreSQL, che dovrebbe essere '**`postgres`**', per cui si potrebbe anche indicare come '`~postgres/`'.

In ogni caso, questa directory è normalmente '`/var/lib/pgsql/`' e tutto ciò che si trova al suo interno appartiene all'utente '**`postgres`**', anche se i permessi per il gruppo e gli altri utenti variano a seconda della circostanza.

Inizialmente, questa directory dovrebbe contenere una serie di file il cui nome inizia per '`pg_*`'. Alcuni di questi sono file di testo, altri sono dei *cataloghi*, ovvero delle tabelle che servono alla gestione del DBMS e non fanno parte delle basi di dati normali. Se per qualche ragione si utilizza l'utente '**`postgres`**', essendo questa la sua directory personale, potrebbero apparire altri file che riguardano la personalizzazione di questo utente ('`.profile`', '`.bash_history`', o altre cose simili, in funzione dei programmi che si utilizzano).

All'interno di questa directory si trova normalmente la sottodirectory '`base/`', da cui si articolano le basi di dati che vengono create di volta in volta: ogni base di dati ottiene una sua sottodirectory ulteriore. Per creare una nuova base di dati, PostgreSQL fa uso di una base di dati di partenza: '**`template1`**'. I file di questa si trovano all'interno di '`base/template1/`'.

324.1.1 Opzioni per la definizione della directory «PGDATA» attraverso la riga di comando

Tutti i programmi che compongono il sistema di PostgreSQL, che hanno la necessità di sapere dove si trovano i dati, oltre al meccanismo della variabile di ambiente '**`PGDATA`**' permettono di indicare tale directory attraverso un'opzione della riga di comando. I programmi più importanti, precisamente '**`postmaster`**' e '**`createdb`**', riconoscono l'opzione '**`-D`**'. Come si può intuire, l'utilizzo di questa opzione, o di un'altra equivalente per gli altri programmi, fa in modo che l'indicazione della variabile '**`PGDATA`**' non abbia effetto.

[1] **PostgreSQL** software libero con licenza speciale

324.1.2 Amministratore

Una particolarità di PostgreSQL sta nella definizione dell'amministratore di questo servizio. In pratica potrebbe trattarsi di una persona diversa dall'amministratore del sistema, l'utente '**root**', corrispondente di solito all'utente '**postgres**'.

Quando la propria distribuzione GNU/Linux è già predisposta per PostgreSQL, l'utente '**postgres**' dovrebbe già essere stato previsto (non importa il numero UID che gli sia stato abbinato), ma quasi sicuramente la parola d'ordine dovrebbe essere «impossibile», come nell'esempio seguente:

```
postgres:!:100:101:PostgreSQL Server:/var/lib/pgsql:/bin/bash
```

Come si vede, il campo della parola d'ordine è occupato da un punto esclamativo che di fatto impedisce l'accesso all'utente '**postgres**'.

A questo punto si pongono due alternative, a seconda che si voglia affidare la gestione del DBMS allo stesso utente '**root**' oppure che si voglia incaricare per questo un altro utente. Nel primo caso non occorrono cambiamenti: l'utente '**root**' può diventare '**postgres**' quando vuole con il comando '**su**';

```
# su postgres
```

nel secondo caso, l'attribuzione di una parola d'ordine all'utente '**postgres**' permetterà a una persona diversa di amministrare il DBMS.

```
# passwd postgres
```

È bene ripetere che la directory iniziale di questo utente fittizio (in questo caso '`/var/lib/pgsql/`') coincide con il punto di inizio della struttura dei dati del DBMS.

324.1.3 Creazione del sistema di basi di dati

La prima volta che si installa PostgreSQL, è molto probabile che venga predisposta automaticamente la directory '`~postgres/`'. Se così non fosse, o se per qualche motivo si dovesse intervenire manualmente, si può utilizzare '**initdb**', che per farlo si avvale di alcune informazioni contenute nella directory definita dalla variabile di ambiente '**PGLIB**', che dovrebbe corrispondere a '`/usr/lib/pgsql/`'.

`initdb` [*opzioni*]

Lo schema sintattico mostra in modo molto semplice l'uso di '**initdb**'. Se si definiscono correttamente le variabili di ambiente '**PGLIB**' e '**PGDATA**', si può fare anche a meno delle opzioni, diversamente diventa necessario dare queste due informazioni attraverso le opzioni della riga di comando.

La directory definita dalla variabile '**PGLIB**', ovvero quella che di solito corrisponde a '`/usr/lib/pgsql/`', serve a '**initdb**' per raggiungere due file: '`global1.bki.source`' e '`local1_template1.bki.source`'. Questi due sono in pratica degli script che servono rispettivamente a generare i file della directory iniziale del sistema di basi di dati, ovvero '`~postgres/*`' ('`/var/lib/pgsql/*`'), e della base di dati '**template1**', corrispondente di solito al contenuto della directory '`~postgres/base/template1/`'. In breve, '**template1**' è lo scheletro utilizzato per la creazione di ogni nuova base di dati.

Prima di avviare '**initdb**', è bene utilizzare l'identità dell'utente amministratore di PostgreSQL:

```
# su postgres
```

Successivamente, avviando '**initdb**', con le indicazioni corrette delle directory corrispondenti alle variabili '**PGLIB**' e '**PGDATA**', si ottengono delle segnalazioni simili a quelle seguenti (si presume che la directory iniziale '**PGDATA**' sia già stata creata e appartenga all'utente '**postgres**').

```
postgres$ initdb --pglib=/usr/lib/pgsql --pgdata=/var/lib/pgsql

We are initializing the database system with username postgres (uid=100).
This user will own all the files and must also own the server process.

Creating Postgres database system directory /var/lib/pgsql/base

Creating template database in /var/lib/pgsql/base/template1

Creating global classes in /var/lib/pgsql/base

Adding template1 database to pg_database...

Vacuuming template1
Creating public pg_user view
Creating view pg_rules
Creating view pg_views
Creating view pg_tables
Creating view pg_indexes
Loading pg_description
```

Alcune opzioni

`--pglib=`*directory_pglib* | `-l` *directory_pglib*

Permette di definire la directory all'interno della quale '**initdb**' deve cercare gli script che servono a ricreare il sistema di basi di dati di PostgreSQL.

`--pgdata=`*directory_pgdata* | `-r` *directory_pgdata*

Stabilisce la directory iniziale del sistema di basi di dati di PostgreSQL che si vuole creare.

`--username=`*amministratore* | `-u` *amministratore*

Questa opzione, permette eventualmente di utilizzare '**initdb**' con i privilegi dell'utente '**root**', definendo in questo modo chi debba essere l'amministratore di PostgreSQL. In generale, questa opzione potrebbe anche non funzionare; per evitare problemi conviene avviare '**initdb**' utilizzando l'identità dell'amministratore PostgreSQL.

`--template` | `-t`

Fa in modo di ricostruire lo scheletro '**template1**', senza intervenire negli altri dati del sistema di basi di dati. Può essere utile se per qualche motivo '**template1**' risulta danneggiato.

324.2 Impostazione cliente-servente e amministrazione

Il DBMS di PostgreSQL si basa su un sistema cliente-servente, in cui, il programma che vuole interagire con una base di dati determinata deve farlo attraverso delle richieste inviate a un servente. In questo modo, il servizio può essere esteso anche attraverso la rete.

L'organizzazione di PostgreSQL prevede la presenza di un demone sempre in ascolto (può trattarsi di un socket di dominio Unix o anche di una porta TCP, che di solito corrisponde al numero 5432). Quando questo riceve una richiesta valida per iniziare una connessione, attiva una copia

del servente vero e proprio (*back-end*), a cui affida la connessione con il cliente. Il demone in ascolto per le richieste di nuove connessioni è '**postmaster**', mentre il servente è '**postgres**'.[2]

Generalmente, il demone '**postmaster**' viene avviato attraverso la procedura di inizializzazione del sistema, in modo indipendente dal supervisore dei servizi di rete. In pratica, di solito si utilizza uno script collocato all'interno di '`/etc/rc.d/init.d/`', o in un'altra collocazione simile, per l'avvio e l'interruzione del servizio.

Durante il funzionamento del sistema, quando alcuni clienti sono connessi, si può osservare una dipendenza del tipo rappresentato dallo schema seguente:

```
...
|
|-postmaster-+-postgres
|            |-postgres
|            `-postgres
...
```

324.2.1 # postmaster

`postmaster` [*opzioni*]

'**postmaster**' è il demone che si occupa di restare in ascolto in attesa di una richiesta di connessione con un servente '**postgres**' (il programma terminale, o *back-end* in questo contesto). Quando riceve questo tipo di richiesta mette in connessione il cliente (o *front-end*) con una nuova copia del servente '**postgres**'.

Per poter compiere il suo lavoro deve essere a conoscenza di alcune notizie essenziali, tra cui in particolare: la collocazione di '**postgres**' (se questo non è in uno dei percorsi della variabile '**PATH**') e la directory da cui si dirama il sistema di file che costituisce il sistema delle varie basi di dati. Queste notizie possono essere predefinite, nella configurazione usata al momento della compilazione dei sorgenti, oppure possono essere indicate attraverso la riga di comando.

'**postmaster**', assieme ai processi da lui controllati (il *back-end*), gestiscono una serie di file che compongono le varie basi di dati del sistema. Trattandosi di un sistema di gestione dei dati molto complesso, è bene evitare di inviare il segnale '**SIGKILL**' (9), perché con questo si provoca la conclusione immediata del processo destinatario e di tutti i suoi discendenti, senza permettere una conclusione corretta. Al contrario, gli altri segnali sono accettabili, come per esempio un '**SIGTERM**' che viene utilizzato in modo predefinito quando si esegue un '**kill**'.

Alcune opzioni

`-D` *directory_dei_dati*

Permette di specificare la directory di inizio della struttura dei dati del DBMS.

`-S`

Specifica che il programma deve funzionare in modo «silenzioso», senza emettere alcuna segnalazione, diventando un processo discendente direttamente da quello iniziale (Init), disassociandosi dalla shell e quindi dal terminale da cui è stato avviato.

Questa opzione viene utilizzata particolarmente per avviare il programma all'interno della procedura di inizializzazione del sistema, quando non sono necessari dei controlli di funzionamento.

[2] Probabilmente, la scelta del nome «postmaster» è un po' infelice, dal momento che potrebbe far pensare all'amministratore del servizio di posta elettronica. Come al solito occorre un po' di attenzione al contesto in cui ci si trova.

`-b` *percorso_del_programma_terminale*

Se il programma terminale, ovvero '**postgres**', non si trova in uno dei percorsi contenuti nella variabile di ambiente '**PATH**', è necessario specificare la sua collocazione (il percorso assoluto) attraverso questa opzione.

`-d` [*livello_di_diagnosi*]

Questa opzione permette di attivare la segnalazione di messaggi diagnostici (*debug*), da parte di '**postmaster**' e da parte dei programmi terminali, a più livelli di dettaglio:

- 1, segnala solo il traffico di connessione;
- 2, o superiore, attiva la segnalazione diagnostica anche nei programmi terminali, oltre ad aggiungere dettagli sul funzionamento di '**postmaster**'.

Di norma, i messaggi diagnostici vengono emessi attraverso lo standard output da parte di '**postmaster**', anche quando si tratta di messaggi provenienti dai programmi terminali. Perché abbia significato usare questa opzione, occorre avviare '**postmaster**' **senza** l'opzione '**-S**'.

`-i`

Abilita le connessioni TCP/IP. Senza l'indicazione di questa opzione, sono ammissibili solo le connessioni locali attraverso socket di dominio Unix (*Unix domain socket*).

`-p` *porta*

Se viene avviato in modo da accettare le connessioni attraverso la rete (l'opzione '**-i**'), specifica una porta di ascolto diversa da quella predefinita (5432).

Esempi

```
# su postgres -c 'postmaster -S -D/var/lib/pgsql'
```

L'utente '**root**', avvia '**postmaster**' dopo essersi trasformato temporaneamente nell'utente '**postgres**' (attraverso '**su**'), facendo in modo che il programma si disassoci dalla shell e dal terminale, diventando un discendente da Init. Attraverso l'opzione '**-D**' si specifica la directory di inizio dei file della base di dati.

```
# su postgres -c 'postmaster -i -S -D/var/lib/pgsql'
```

Come nell'esempio precedente, specificando che si vuole consentire, in modo preliminare, l'accesso attraverso la rete.[3]

```
# su postgres -c 'nohup postmaster -D/var/lib/pgsql ↩
↪> /var/log/pglog 2>&1 &'
```

L'utente '**root**', avvia '**postmaster**' in modo simile al precedente, dove in particolare viene diretto lo standard output all'interno di un file, per motivi diagnostici. Si osservi l'utilizzo di '**nohup**' per evitare l'interruzione del funzionamento di '**postmaster**' all'uscita del programma '**su**'.

```
# su postgres -c 'nohup postmaster -D/var/lib/pgsql -d 1 ↩
↪> /var/log/pglog 2>&1 &'
```

Come nell'esempio precedente, con l'attivazione del primo livello diagnostico nei messaggi emessi.

[3] Per consentire in pratica l'accesso attraverso la rete, occorre anche intervenire all'interno del file di configurazione '`~postgres/pg_hda.conf`'.

324.2.2 Localizzazione

A partire dalla versione 6.4 di PostgreSQL, inizia l'introduzione di un sistema di gestione delle localizzazioni. La sua attivazione dipende dalle opzioni che vengono definite in fase di compilazione, per cui potrebbe anche succedere che la propria distribuzione GNU/Linux disponga di una versione di PostgreSQL che non è in grado di gestire la localizzazione.

La localizzazione va applicata al servente, ovvero al sistema di gestione dei dati, mentre ciò non serve dalla parte del cliente. Questa è una situazione un po' strana rispetto al solito, dove ogni utente configura per sé il proprio ambiente. Infatti, la scelta della localizzazione dei dati, deve essere fatta al livello della base di dati, senza poter essere cambiata a piacimento, a seconda dei punti di vista.

Di conseguenza, la configurazione delle variabili '**`LC_*`**', o eventualmente di '**`LANG`**', deve avvenire per l'ambiente riferito al funzionamento di '**`postmaster`**', per cui occorre preparare uno script apposito.

```
#!/bin/sh

LANG=it_IT.ISO-8859-1
# LC_CTYPE=it_IT.ISO-8859-1
# LC_COLLATE=it_IT.ISO-8859-1
# LC_MONETARY=it_IT.ISO-8859-1
export LANG
# export LC_CTYPE LC_COLLATE LC_MONETARY

/usr/bin/postmaster -i -S -D/var/lib/pgsql
```

Lo script che si vede sopra, serve a definire la variabile di ambiente '**`LANG`**', a esportarla e ad avviare '**`postmaster`**'. Questo script deve essere avviato dalla procedura di inizializzazione del sistema, all'interno della quale sarà utilizzato presumibilmente '**`su`**', in modo da attribuire l'identità dell'utente amministratore di PostgreSQL. Se si usa un sistema di script per l'avvio o la conclusione dei servizi, cosa che di solito si colloca nella directory '`/etc/init.d/`', o '`/etc/rc.d/init.d/`', potrebbe essere necessario intervenire su quello che si occupa di avviare '**`postmaster`**'.

```
#!/bin/sh
case "$1" in
  start)
        echo -n "Avvio del servizio PostgreSQL: "
        su -l postgres -c '/usr/bin/postmaster -i -S -D/var/lib/pgsql'
        echo
        ;;
  stop)
        echo -n "Disattivazione del servizio PostgreSQL: "
        killall postmaster
        echo
        ;;
  *)
        echo "Utilizzo: postgresql {start|stop}"
        exit 1
esac
```

Quello che si vede sopra, è lo scheletro della struttura '**`case`**' tipica di un tale script. Volendo modificare la localizzazione predefinita in fase di compilazione, occorre lo script mostrato prima. Si suppone che lo script con il quale si modificano le variabili di localizzazione e si avvia '**`postmaster`**', sia '`/usr/bin/avvia_postmaster`'; la modifica da apportare all'esempio appena visto è quella seguente:

```
...
case "$1" in
  start)
```

```
        echo -n "Avvio del servizio PostgreSQL: "
        # su -l postgres -c '/usr/bin/postmaster -i -S -D/var/lib/pgsql'
        su -l postgres -c '/usr/bin/avvia_postmaster'
        echo
        ;;
...
```

Oltre alla localizzazione attraverso le variabili di ambiente tradizionali, si può intervenire sulla variabile '**PGDATESTYLE**', il cui scopo è quello di definire la forma di visualizzazione delle date. La tabella 324.1 elenca le parole chiave che si possono assegnare a questa variabile e l'effetto che ne deriva.

Tabella 324.1. Elenco dei formati di data gestibili con PostgreSQL.

Stile	Descrizione	Esempio
ISO	ISO 8601	1999-12-31
SQL	Tipo tradizionale	12/31/1999
German		31.12.1999

Probabilmente, la cosa migliore è utilizzare il formato '**ISO**', che potrebbe anche diventare quello predefinito nelle prossime versioni di PostgreSQL. Volendo estendere lo script per l'avvio di '**postmaster**', presentato all'inizio, basta aggiungere l'impostazione della variabile '**PGDATESTYLE**':

```
#!/bin/sh

LANG=it_IT.ISO-8859-1
# LC_CTYPE=it_IT.ISO-8859-1
# LC_COLLATE=it_IT.ISO-8859-1
# LC_MONETARY=it_IT.ISO-8859-1
export LANG
# export LC_CTYPE LC_COLLATE LC_MONETARY
PGDATESTYLE=ISO
export PGDATESTYLE

/usr/bin/postmaster -i -S -D/var/lib/pgsql
```

324.2.3 Organizzazione degli utenti e delle basi di dati

Per fare in modo che gli utenti possano accedere al DBMS, occorre che siano stati registrati all'interno del sistema di PostgreSQL stesso. In pratica, può trattarsi solo di utenti già riconosciuti nel sistema operativo, che vengono aggiunti e accettati anche da PostgreSQL. Per l'inserimento di questi utenti si utilizza '**createuser**', come nell'esempio seguente:

```
# su postgres[ Invio ]

postgres$ createuser[ Invio ]

Enter name of user to add---> daniele[ Invio ]

Enter user's postgres ID or RETURN to use unix user ID: 500 -> [ Invio ]
```

In tal modo è stato definito l'inserimento dell'utente '**daniele**', confermando il suo numero UID.

```
Is user "daniele" allowed to create databases (y/n) y[ Invio ]
```

All'utente '**daniele**' è stato concesso di creare delle nuove basi di dati.

```
Is user "daniele" allowed to add users? (y/n) n[ Invio ]
```

All'utente non viene concesso di aggiungere altri utenti.

```
createuser: daniele was successfully added
```

Da questo esempio si può comprendere quali siano le possibilità di attribuzione di privilegi ai vari utenti del sistema DBMS. In particolare, è opportuno osservare che ogni base di dati appartiene all'utente che lo ha creato, il quale diventa il suo amministratore particolare (per la precisione il DBA).

L'eliminazione di un utente PostgreSQL avviene in modo simile attraverso '**destroyuser**', come nell'esempio seguente:

```
# su postgres[ Invio ]

postgres$ destroyuser[ Invio ]

Enter name of user to delete ---> daniele[ Invio ]

destroyuser: delete of user daniele was successful.
```

> L'eliminazione di un utente PostgreSQL comporta anche l'eliminazione delle basi di dati a lui appartenenti.

> Le informazioni sugli utenti autorizzati a gestire in qualunque modo il sistema di basi di dati sono archiviate nel file '`~postgres/pg_shadow`', visibile anche attraverso la vista definita dal file '`~postgres/pg_user`'. È utile sapere questo per comprendere il significato dei messaggi di errore, quando fanno riferimento a questo file.

324.2.4 Controllo diagnostico

Inizialmente, l'utilizzo di PostgreSQL si può dimostrare poco intuitivo, soprattutto per ciò che riguarda le segnalazioni di errore, spesso troppo poco esplicite. Per permettere di avere una visione un po' più chiara di ciò che accade, sarebbe bene fare in modo che '**postmaster**' produca dei messaggi diagnostici, possibilmente diretti a un file o a una console virtuale inutilizzata.

Nella sezione in cui si descrive il funzionamento di '**postmaster**' appaiono alcuni esempi di avvio di questo programma, in modo da generare e conservare queste informazioni diagnostiche. L'esempio seguente, in particolare, avvia '**postmaster**' in modo manuale e, oltre a conservare le informazioni diagnostiche in un file, le visualizza continuamente attraverso una console virtuale inutilizzata (l'ottava).

```
# su postgres[ Invio ]

$ nohup postmaster -D/var/lib/pgsql -d 1 > /var/log/pglog 2>&1 &[ Invio ]

$ exit[ Invio ]

# nohup tail -f /var/lib/pgsql > /dev/tty8 &[ Invio ]
```

324.3 Accesso e autenticazione

L'accesso alle basi di dati viene consentito attraverso un sistema di autenticazione. I sistemi di autenticazione consentiti possono essere diversi e dipendono dalla configurazione di PostgreSQL fatta all'atto della compilazione dei sorgenti.

Il file di configurazione '`pg_hba.conf`' (*Host-based authentication*), che si trova della directory iniziale dell'utente '**`postgres`**', cioè l'inizio della struttura delle basi di dati, serve per controllare il sistema di autenticazione una volta installato PostgreSQL.

L'autenticazione degli utenti può avvenire in modo incondizionato ('**`trust`**'), cosa che si fa di solito quando chi accede è un utente del sistema presso cui è in funzione PostgreSQL stesso; in pratica ci si fida del sistema di controllo fatto dal sistema operativo.

L'autenticazione può essere semplicemente disabilitata, nel senso di impedire qualunque accesso incondizionatamente. Questo può servire per impedire l'accesso da parte di un certo gruppo di nodi.

L'accesso può essere controllato attraverso l'abbinamento di una parola d'ordine agli utenti di PostgreSQL. Queste parole d'ordine possono essere conservate in un file di testo con una struttura simile a quella di '`/etc/passwd`', oppure nel file '`~postgres/pg_shadow`', che in pratica è una tabella (questo particolare verrà ripreso in seguito).

Inoltre, l'autenticazione può avvenire attraverso un sistema Kerberos, oppure attraverso il protocollo IDENT (capitolo 185). In questo ultimo caso, ci si fida di quanto riportato dal sistema remoto il quale conferma o meno che la connessione appartenga a quell'utente che si sta connettendo.

324.3.1 ~postgres/pg_hba.conf

Il file '`~postgres/pg_hba.conf`' permette di definire quali nodi possono accedere al servizio DBMS di PostgreSQL, eventualmente stabilendo anche un abbinamento specifico tra basi di dati e nodi di rete.

Le righe vuote e il testo preceduto dal simbolo '`#`' vengono ignorati. I record (cioè le righe contenenti le direttive del file in questione), sono suddivisi in campi separati da spazi o caratteri di tabulazione. Il formato può essere riassunto nei due modelli sintattici seguenti:

`local` *base_di_dati autenticazione_utente* [*mappa*]
`host` *base_di_dati indirizzo_ip maschera_degli_indirizzi autenticazione_utente* [*mappa*]

Nel primo caso si intendono controllare gli accessi provenienti da clienti avviati nello stesso sistema locale, utilizzando un socket di dominio Unix; nel secondo si fa riferimento ad accessi attraverso la rete (connessioni TCP).

- Il secondo campo del record serve a indicare il nome di una base di dati per la quale autorizzare l'accesso; in alternativa si può usare la parola chiave '**`all`**', in modo da specificare tutte le basi di dati in una sola volta.
- I campi *indirizzo_ip* e il successivo, *maschera_degli_indirizzi* , rappresentano un gruppo di indirizzi di nodi che hanno diritto di accedere a quella base di dati determinata.
- Il campo *autenticazione_utente* rappresenta il tipo di autenticazione attraverso una parola chiave. Le più comuni sono:
  - '**`trust`**' -- l'autenticazione non ha luogo e si accetta il nome fornito dall'utente senza alcuna verifica.

  - '**reject**' -- la connessione viene rifiutata in ogni caso.
  - '**password**' -- viene richiesta una parola d'ordine riferita all'utente, verificandola in base al contenuto di un file indicato nel campo successivo, oppure in base a quanto riportato dal catalogo '`~postgres/pg_shadow`'.
  - '**crypt**' -- viene richiesta una parola d'ordine riferita all'utente, verificandola in base al contenuto di '`~postgres/pg_shadow`'. La differenza più importante rispetto a '**password**' sta nel fatto che in quel caso la parola d'ordine viene trasmessa in chiaro, mentre con '**crypt**' no.
  - '**ident**' -- l'autenticazione avviene attraverso il protocollo IDENT (capitolo 185), demandando il riconoscimento al sistema remoto.[4]

- L'ultimo campo dipende dal penultimo. Nel caso di autenticazione '**ident**', si utilizza solitamente la parola chiave '**sameuser**' per indicare a PostgreSQL che i nomi usati dagli utenti nei sistemi remoti da cui possono accedere, coincidono con quelli predisposti per la gestione del DBMS. Nel caso di autenticazione '**password**' rappresenta il nome del file di testo contenente le parole d'ordine.[5]

Perché il sistema possa funzionare correttamente, sono sempre presenti almeno i record seguenti:

```
# tipo  database        IP              maschera                autorizz.
#
local        all                                                 trust
host         all         127.0.0.1     255.255.255.255           trust
```

Ciò consente l'accesso senza altre misure di sicurezza a tutti i clienti che accedono dallo stesso sistema locale attraverso un socket di dominio Unix, e agli utenti dello stesso nodo locale (*localhost*), a tutte le basi di dati.

L'esempio seguente permette l'accesso da parte di utenti provenienti dalla rete locale 192.168.∗.∗, alla base di dati '**nostro_db**', affidando il compito di riconoscimento al sistema remoto da cui avviene la connessione e utilizzando il nome dell'utente, fornito in questo modo, come nome di utente PostgreSQL.

```
# tipo  database        IP              maschera        autorizz.
#
host    nostro_db       192.168.0.0     255.255.0.0     ident   sameuser
```

L'esempio seguente, è simile al precedente, con la differenza che gli accessi dalla rete indicata richiedono una parola d'ordine, che PostgreSQL conserva nel file di testo '`~postgres/passwd`' (il nome indicato nell'ultimo campo).

```
# tipo  database        IP              maschera        autorizz.
#
host    nostro_db       192.168.0.0     255.255.0.0     password  passwd
```

Questo file di configurazione viene fornito già con alcuni esempi commentati.

324.3.2 Gestione delle parole d'ordine in chiaro

Con il sistema di autenticazione definito dalla parola chiave '**password**' è possibile utilizzare un file di testo simile a '`/etc/passwd`' o a '`/etc/shadow`' per annotare gli utenti PostgreSQL e le parole d'ordine cifrate relative. Per esempio, se nel file '`~postgres/pg_hba.conf`' compare il record

[4] Si intende che questo metodo sia anche molto poco sicuro.

[5] L'autenticazione IDENT prevede anche l'uso di un file di mappa aggiuntivo, che viene preso in considerazione quando al posto della parola chiave '**sameuser**' si indica qualcosa d'altro. Tuttavia, la documentazione sul modo in cui debba essere predisposto questo file non è disponibile allo stato attuale.

```
host    nostro_db       192.168.0.0     255.255.0.0     password  utenti
```

gli utenti che accedono attraverso un cliente avviato dai nodi della sottorete 192.168.∗.∗ devono identificarsi attraverso l'indicazione di una parola d'ordine che PostgreSQL può trovare nel file di testo '`~postgres/utenti`'. Questo file potrebbe essere simile a quello seguente:

```
tizio:wsLHjp.FutW0s
caio:a6%i/.45w2q4
```

Se questo file dovesse contenere dei campi aggiuntivi (separati con i soliti due punti), questi verrebbero semplicemente ignorati.

> Quando il cliente deve accedere utilizzando questo tipo di autenticazione, deve presentarsi con il nominativo-utente e la parola d'ordine. Quando si usa il programma '**`psql`**' che verrà descritto in seguito, occorre specificare l'opzione '**`-u`**'.

La parola d'ordine cifrata che si colloca nel secondo campo del record di questo file è ottenuta con la solita funzione di sistema '**`crypt()`**'. Per inserire facilmente un utente, o per cambiare la parola d'ordine di un utente registrato precedentemente, si utilizza il programma '**`pg_passwd`**', indicando semplicemente in quale file intervenire.

```
pg_passwd file
```

L'utilizzo è banale, come si vede dall'esempio seguente in cui si aggiunge l'utente '**`semproni`**' (è importante ricordare di operare in qualità di utente '**`postgres`**').

```
# cd ~postgres[ Invio ]

# su postgres[ Invio ]

postgres:~$ pg_passwd utenti[ Invio ]

Username: semproni[ Invio ]

New password: ******[ Invio ]

Re-enter new password: ******[ Invio ]
```

324.4 Configurazione nella distribuzione GNU/Linux Debian

La distribuzione GNU/Linux Debian è molto attenta alla coerenza dei pacchetti che si installano; pertanto, nel caso di PostgreSQL, può essere controllato tutto a partire dai file che si trovano nella directory '`/etc/postgresql/`'. In particolare, si trova in questa directory il file '`pg_hba.conf`' che è già stato descritto in precedenza; inoltre, si trova un file aggiuntivo che viene interpretato dallo script della procedura di inizializzazione del sistema che si occupa di avviare e di arrestare il servizio. Si tratta del file '`/etc/postgresql/postmaster.init`', attraverso il quale si possono controllare tante piccole cose, che altrimenti andrebbero controllate attraverso le opzioni della riga di comando del demone relativo.

```
# /etc/postgresql/postmaster.init
#
# Copyright (c) Oliver Elphick 1997
# Part of the Debian package, postgresql. The Debian packaging is
# licensed under GPL v.2
#
```

```
# This is the configurable initialisation of the postgresql package
# The defaults are shown, but are commented out.
#
POSTGRES_HOME=‘grep ’^postgres:’ /etc/passwd | awk -F: ’{print $6}’‘
if [ -z "$POSTGRES_HOME" ]
then
    POSTGRES_HOME=/var/postgres
fi

# Where to find the PostgreSQL database files, including those that
# define PostgreSQL users and permissions.
# POSTGRES_DATA=/var/postgres/data

# Where to send logging and debugging traces
# POSTGRES_LOG=/var/log/postgres.log

# The number of shared-memory buffers the postmaster is
# to allocate for backend server processes.  Each buffer is 8Kb.
# PGBUFFERS=64

# Debugging level at which the backend servers are to operate.
# 1: trace connection traffic only; >=2: turn on debugging in the backends
# giving more information according to the debug level. Debug logs are
# sent to $POSTGRES_LOG
# PGDEBUG=0

# Whether to echo queries to the debug log: yes/no
# PGECHO=no

# Whether to disable the fsync() call after each transaction. (If fsync() is
# disabled, performance will improve at the cost of an increased risk of data
# corruption in the event of power or other hardware failure.): yes/no
# PGFSYNC=yes

# How to present dates to the frontend. The choices are American (mm-dd-yyyy)
# or European (dd-mm-yyyy)
# PGDATESTYLE=European

# How much memory to use for internal sorts before resorting to the disk.
# This value is in kilobytes.
# PGSORTMEM=512

# Whether to print timing and other statistics after each query: yes/no
# PGSTATS=no

# Whether to allow connections through TCP/IP as well as through Unix
# sockets: yes/no.
# By default, for greater security, we do not allow TCP/IP access.
# This means that only users on this machine can access the database.
PGALLOWTCPIP=yes

# The Internet TCP port on which postmaster is to listen for connections
# from frontend applications.
# PGPORT=5432

# Locale setting for the postmaster and backend to use: this is not
# necessary for USA users, but most others will probably want to set it; it
# controls things like the format of numbers and dates.
# for example, use ‘LANG=en_GB’ for British English.
LANG=it_IT
```

Quello che si vede è l’esempio del file predefinito con alcuni ritocchi per adattarlo alle particolarità locali. È importante sottolineare che per motivi di sicurezza, l’accesso tramite la rete viene impedito inizialmente, per cui occorre abilitare la cosa in modo esplicito, attraverso la direttiva ‘**`PGALLOWTCPIP=yes`**’, come si vede dall’esempio stesso.

324.5 Gestione delle basi di dati

Per poter gestire una base di dati occorre prima crearla. Ciò si ottiene normalmente attraverso lo script '**createdb**', avviato con i privilegi adatti, cioè quelli di un utente a cui ciò è consentito. Nello stesso modo, attraverso lo script '**dropdb**', si può eliminare un'intera base di dati.

> PostgreSQL non distingue tra lettere maiuscole e minuscole quando si tratta di nominare le basi di dati, le relazioni (le tabelle o gli oggetti a seconda della definizione che si preferisce utilizzare) e gli elementi delle relazioni. Tuttavia, in certi casi si verificano degli errori inspiegabili dovuti alla scelta dei nomi che in generale conviene indicare sempre solo con lettere minuscole.

324.5.1 Creazione di una base di dati

La creazione di una base di dati è in pratica la creazione di una serie di file all'interno di una directory con lo stesso nome usato per identificare la base di dati stessa. Questa operazione ha luogo utilizzando una struttura di partenza già predisposta: di solito si tratta di '**template1**'.

Le directory delle basi di dati si articolano a partire da '`~postgres/base/`'. Quando si crea l'ipotetica base di dati '**mio_db**', ciò che si ottiene in pratica è la copia della directory '`~postgres/base/template1/`' in '`~postgres/base/mio_db/`'.[6]

Come già accennato, una base di dati può essere creata solo da un utente autorizzato precedentemente per questo scopo. Di solito si utilizza lo script '**createdb**', come nell'esempio seguente in cui si crea la base di dati '**mio_db**'.

```
$ createdb mio_db
```

L'utente che ha creato una base di dati è automaticamente il suo amministratore, ovvero colui che può decidere eventualmente di eliminarla.

PostgreSQL pone dei limiti nella scelta dei nomi delle basi di dati. Non possono superare i 16 caratteri e il primo di questi deve essere alfabetico, oppure può essere un trattino basso.[7]

Se l'utente che tenta di creare una base di dati non è autorizzato per questo, quello che si ottiene è un messaggio di errore del tipo seguente:

```
Connection to database 'template1' failed.
FATAL 1:SetUserId: user "tizio" is not in "pg_user"
createdb: database creation failed on mio_db.
```

324.5.2 Eliminazione di una base di dati

L'amministratore di una base di dati, generalmente colui che la ha creata, è la persona che può anche eliminarla. Nell'esempio seguente si elimina la base di dati '**mio_db**'.

```
$ dropdb mio_db
```

[6] In ogni caso, la copia da sola non basta. Perché una base di dati sia riconosciuta come tale occorre che questa sia stata annotata nel file '`~postgres/pg_database`'.

[7] La dimensione massima dei nomi dipende dal modo in cui sono stati compilati i sorgenti o dalle caratteristiche della piattaforma. Il limite di 16 caratteri è sufficientemente basso da andare bene in ogni circostanza.

324.6 Accesso a una base di dati

L'accesso a una base di dati avviene attraverso un cliente, ovvero un programma frontale, o *front-end*, secondo la documentazione di PostgreSQL. Questo si avvale generalmente della libreria LibPQ. PostgreSQL fornisce un programma cliente standard, '**psql**', che si comporta come una sorta di shell tra l'utente e la base di dati stessa.[8]

Il programma '**psql**' permette un utilizzo interattivo attraverso una serie di comandi impartiti dall'utente su una riga di comando; oppure può essere avviato in modo da eseguire il contenuto di un file o di un solo comando fornito tra gli argomenti. Per quanto riguarda l'utilizzo interattivo, il modo più semplice per avviarlo è quello che si vede nell'esempio seguente, dove si indica semplicemente il nome della base di dati sulla quale intervenire.

```
$ psql mio_db[ Invio ]

Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: mio_db

mio_db=>_
```

Da questo momento si possono inserire le istruzioni SQL per la base di dati selezionata, in questo caso '**mio_db**', oppure si possono inserire dei comandi specifici di '**psql**'. Questi ultimi si notano perché sono composti da una barra obliqua inversa ('\'), seguita da un carattere.

Il comando interno di '**psql**' più importante è '**\h**' che permette di visualizzare una guida rapida alle istruzioni SQL che possono essere utilizzate.

```
=> \h[ Invio ]

type \h <cmd> where <cmd> is one of the following:
    abort                   abort transaction       alter table
    begin                   begin transaction       begin work
    cluster                 close                   commit
...
type \h * for a complete description of all commands
```

Nello stesso modo, il comando '**\?**' fornisce un riepilogo dei comandi interni di '**psql**'.

```
=> \?[ Invio ]

 \?           -- help
 \a           -- toggle field-alignment (currenty on)
 \C [<captn>] -- set html3 caption (currently '')
...
```

Tutto ciò che '**psql**' non riesce a interpretare come un suo comando interno viene trattato come un'istruzione SQL. Dal momento che queste istruzioni possono richiedere più righe, è necessario informare '**psql**' della conclusione di queste, per permettergli di analizzarle e inviarle al servente. Queste istruzioni possono essere terminate con un punto e virgola ('**;**'), oppure con il comando '**\g**'.

[8] Il programma cliente tipico, dovrebbe riconoscere le variabili di ambiente '**PGHOST**' e '**PGPORT**'. La prima serve a stabilire l'indirizzo o il nome di dominio del servente, indicando implicitamente che la connessione avviene attraverso una connessione TCP e non con un socket di dominio Unix; la seconda specifica il numero della porta, ammesso che si voglia utilizzare un numero diverso da 5432. L'uso di queste variabili non è indispensabile, ma serve solo per non dover specificare queste informazioni attraverso opzioni della riga di comando.

Si può osservare, utilizzando '**psql**', che l'invito mostrato cambia leggermente a seconda del contesto: inizialmente appare nella forma '**=>**', mentre quando è in corso l'inserimento di un'istruzione SQL non ancora terminata si trasforma in '**->**'. Il comando '**\g**' viene usato prevalentemente in questa situazione.

```
-> \g[ Invio ]
```

Le istruzioni SQL possono anche essere raccolte in un file di testo normale. In tal caso si può utilizzare il comando '**\i**' per fare in modo che '**psql**' interpreti il suo contenuto, come nell'esempio seguente, dove il file in questione è '`mio_file.sql`'.

```
=> \i mio_file.sql[ Invio ]
```

Nel momento in cui si utilizza questa possibilità (quella di scrivere le istruzioni SQL in un file facendo in modo che poi questo venga letto e interpretato), diventa utile il poter annotare dei commenti. Questi sono iniziati da una sequenza di due trattini ('**--**'): tutto quello che vi appare dopo viene ignorato.

La conclusione del funzionamento di '**psql**' si ottiene con il comando '**\q**'.

```
=> \q[ Invio ]
```

324.6.1 $ psql

`psql` [*opzioni*] [*base_di_dati*]

'**psql**' è un programma frontale (*front-end*) interattivo per l'invio di istruzioni SQL e l'emissione del risultato corrispondente. Si tratta di un cliente come gli altri, di conseguenza richiede la presenza di '**postmaster**' per instaurare una connessione con una copia del servente '**postgres**'.

'**psql**' può funzionare in modo interattivo, come già accennato, oppure può eseguire le istruzioni contenute in un file. Questo può essere fornito attraverso l'opzione '**-f**', oppure può provenire dallo standard input, attraverso una pipeline.

'**psql**' può funzionare solo in abbinamento a una base di dati determinata. In questo senso, se non viene indicato il nome di una base di dati nella riga di comando, '**psql**' tenta di utilizzarne una con lo stesso nome dell'utente. Per la precisione, si fa riferimento alla variabile di ambiente '**USER**'.[9]

Alcune opzioni

`-c` *istruzione_SQL*

Permette di fornire un'istruzione SQL già nella riga di comando, ottenendone il risultato attraverso lo standard output e facendo terminare subito dopo l'esecuzione di '**psql**'. Questa opzione viene usata particolarmente in abbinamento a '**-q**'.

`-d` *base_di_dati*

Permette di indicare il nome della base di dati da utilizzare. Può essere utile quando per qualche motivo potrebbe essere ambigua l'indicazione del suo nome come ultimo argomento.

`-f` *file_di_istruzioni*

[9] Questo dettaglio dovrebbe permettere di comprendere il significato della segnalazione di errore che si ottiene se si tenta di avviare '**psql**' senza indicare una base di dati, quando non ne esiste una con lo stesso nome dell'utente.

Permette di fornire a '**psql**' un file da interpretare contenente le istruzioni SQL (oltre agli eventuali comandi specifici di '**psql**'), senza avviare così una sessione di lavoro interattiva.

`-h` *host*

Permette di specificare il nodo a cui connettersi per l'interrogazione del servente PostgreSQL.

`-H`

Fa in modo che l'emissione di tabelle avvenga utilizzando il formato HTML 3.0. In pratica, ciò è utile per costruire un risultato da leggere attraverso un navigatore *web*.

`-o` *file_output*

Fa in modo che tutto l'output venga inviato nel file specificato dall'argomento.

`-p` *porta*

Nel caso in cui '**postmaster**' sia in ascolto su una porta TCP diversa dal numero 5432 (corrispondente al valore predefinito), si può specificare con questa opzione il numero corretto da utilizzare.

`-q`

Fa sì che '**psql**' funzioni in modo «silenzioso», limitandosi all'emissione pura e semplice di quanto generato dalle istruzioni impartite. Questa opzione è utile quando si utilizza '**psql**' all'interno di script che devono occuparsi di rielaborare il risultato ottenuto.

`-t`

Disattiva l'emissione dei nomi delle colonne. Questa opzione viene utilizzata particolarmente in abbinamento con '**-c**' o '**-q**'.

`-T` *opzioni_tabelle_html*

Questa opzione viene utilizzata in abbinamento con '**-H**', per definire le opzioni HTML delle tabelle che si generano. In pratica, si tratta di ciò che può essere inserito all'interno del marcatore di apertura della tabella: '**<table …>**'.

`-u`

Fa in modo che '**psql**' richieda il nominativo-utente e la parola d'ordine all'utente, prima di tentare la connessione. L'uso di questa opzione è indispensabile quando il servente impone una forma di autenticazione definita attraverso la parola chiave '**password**'.

Alcuni comandi

Oltre alle istruzioni SQL, '**psql**' riconosce dei comandi, alcuni dei quali vengono descritti di seguito.

`\h` [*comando*]

L'opzione '**\h**' usata da sola, elenca le istruzioni SQL che possono essere utilizzate. Se viene indicato il nome di una di queste, viene mostrata in breve la sintassi relativa.

`\?`

Elenca i comandi interni di '**psql**', cioè quelli che iniziano con una barra obliqua inversa ('****').

`\l`

Elenca tutte le basi di dati presenti nel servente. Ciò che si ottiene è una tabella contenente rispettivamente: i nomi delle basi di dati, i numeri UID dei rispettivi amministratori (gli utenti che li hanno creati) e il nome della directory in cui sono collocati fisicamente.

`\connect` *base_di_dati* [*nome_utente*]

Chiude la connessione con la base di dati in uso precedentemente e tenta di accedere a quella indicata. Se il sistema di autenticazione lo consente, si può specificare anche il nome

dell'utente con cui si intende operare sulla nuova base di dati. Generalmente, ciò dovrebbe essere impedito.[10]

`\d [tabella]`

L'opzione '**\d**' usata da sola, elenca le tabelle contenute nella base di dati, altrimenti, se viene indicato il nome di una di queste tabelle, si ottiene l'elenco delle colonne. Se si utilizza il comando '**\d ***', si ottiene l'elenco di tutte le tabelle con le informazioni su tutte le colonne rispettive.

`\i file`

Con questa opzione si fa in modo che '**psql**' esegua di seguito tutte le istruzioni contenute nel file indicato come argomento.

`\q`

Termina il funzionamento di '**psql**'.

Codici di uscita

Il programma '**psql**' può restituire i valori seguenti:

- 0 se tutte le istruzioni sono state eseguite senza errori;
- 1 se si sono verificati errori;
- 2 se è intervenuta una disconnessione da parte del servente sottostante.

Esempi

```
$ psql mio_db
```

Cerca di connettersi con la base di dati '**mio_db**' nel nodo locale, utilizzando il meccanismo del socket di dominio Unix.

```
$ psql -d mio_db
```

Esattamente come nell'esempio precedente, con l'uso dell'opzione '**-d**' che serve a evitare ambiguità sul fatto che '**mio_db**' sia il nome della base di dati.

```
$ psql -u -d mio_db
```

Come nell'esempio precedente, ma fa in modo che '**psql**' chieda all'utente il nominativo e la parola d'ordine da usare per collegarsi. È necessario usare questa opzione quando il servizio a cui ci si connette richiede un'autenticazione basata sull'uso di parole d'ordine.

```
$ psql -u -h dinkel.brot.dg -d mio_db
```

Come nell'esempio precedente, ma questa volta l'accesso viene fatto a una base di dati con lo stesso nome presso il nodo *dinkel.brot.dg*.

```
$ psql -f istruzioni.sql -d mio_db
```

Cerca di connettersi con la base di dati '**mio_db**' nel nodo locale, utilizzando il meccanismo del socket di dominio Unix, quindi esegue le istruzioni contenute nel file '`istruzioni.sql`'.

[10] Se si utilizza un'autenticazione basata sul file '`pg_hba.conf`', l'autenticazione di tipo '**trust**' consente questo cambiamento di identificazione, altrimenti, il tipo '**ident**' lo impedisce. La configurazione normale prevede che il nodo locale (127.0.0.1) possa accedere con un'autenticazione di tipo '**trust**', cosa che permette di cambiare il nome dell'utente in questo comando.

324.6.2 Variabile PAGER

'**psql**' è sensibile alla presenza o meno della variabile di ambiente '**PAGER**'. Se questa esiste e non è vuota, '**psql**' userà il programma indicato al suo interno per controllare l'emissione dell'output generato. Per esempio, se contiene '**less**', come si vede nell'esempio seguente che fa riferimento a una shell compatibile con quella di Bourne,

```
PAGER=less
export PAGER
```

si fa in modo che l'output troppo lungo venga controllato da '**less**'. Per eliminare l'impostazione di questa variabile, in modo da ritornare allo stato predefinito, basta annullare il contenuto della variabile nel modo seguente:

```
PAGER=
export PAGER
```

324.7 Manutenzione delle basi di dati

Un problema comune dei DBMS è quello della riorganizzazione periodica dei dati, in modo da semplificare e accelerare le elaborazioni successive. Nei sistemi più semplici si parla a volte di «ricostruzione indici», o di qualcosa del genere. Nel caso di PostgreSQL, si utilizza un comando specifico che è estraneo all'SQL standard: '**VACUUM**'.[11]

VACUUM [VERBOSE] [ANALYZE] [*nome_tabella*]
VACUUM [VERBOSE] ANALYZE [*nome_tabella* [(*colonna_1* [, … *colonna_N*])]]

L'operazione di pulizia si riferisce alla base di dati aperta in quel momento. L'opzione '**VERBOSE**' permette di ottenere i dettagli sull'esecuzione dell'operazione; '**ANALYZE**' serve invece per indicare specificatamente una tabella, o addirittura solo alcune colonne di una tabella.

Anche se non si tratta di un comando SQL standard, per PostgreSQL è importante che venga eseguita periodicamente una ripulitura con il comando '**VACUUM**', eventualmente attraverso uno script simile a quello seguente, da avviare per mezzo del sistema Cron.

```
#!/bin/sh
su postgres -c "psql $1 -c 'VACUUM'"
```

In pratica, richiamando questo script con i privilegi dell'utente '**root**', indicando come argomento il nome della base di dati (viene inserito al posto di '**$1**' dalla shell), si ottiene di avviare il comando '**VACUUM**' attraverso '**psql**'.

Per riuscire a fare il lavoro in serie per tutte le basi di dati, si potrebbe scrivere uno script più complesso, come quello seguente. In questo caso, lo script deve essere avviato con i privilegi dell'utente '**postgres**'.

```
#!/bin/sh

BASI_DATI=‘psql template1 -t -c "SELECT datname from pg_database"‘

echo "Procedimento di ripulitura e sistemazione delle basi di dati"
echo "di PostgreSQL."
echo "Se l’operazione dovesse essere interrotta accidentalmente,"
echo "potrebbe essere necessaria l’eliminazione del file pg_vlock"
echo "contenuto nella directory della <ttid>base di dati</ttid> relativa."

for BASE_DATI in $BASI_DATI
```

[11] Per comprendere bene il contenuto di questa sezione, può essere necessaria la lettura del prossimo capitolo. Queste informazioni sono collocate qui soltanto per una questione di ordine logico nella posizione delle stesse.

```
do
    echo -n "$BASE_DATI: "
    psql $BASE_DATI -c "VACUUM"
done
```

In breve, si utilizza la prima volta '**psql**' in modo da aprire la base di dati '**template1**' (quella fondamentale, che permette di intervenire sui cataloghi di sistema), accedendo al catalogo '**pg_database**' per leggere la colonna contenente i nomi delle basi di dati. In particolare, l'opzione '**-t**' serve a evitare di inserire il nome della colonna stessa. L'elenco che si ottiene viene inserito nella variabile di ambiente '**BASI_DATI**', che in seguito viene scandita da un ciclo '**for**', all'interno del quale si utilizza '**psql**' per ripulire ogni singola base di dati.

324.8 Maneggiare i file delle basi di dati

All'inizio del capitolo si è accennato alla collocazione normale delle directory e dei file che compongono le basi di dati. Chi amministra il sistema di elaborazione che ospita PostgreSQL e le basi di dati, deve avere almeno un'idea di come maneggiare questi file. Per esempio deve sapere come comportarsi per le copie di sicurezza, soprattutto come ripristinarle.

Per comodità, la directory da cui si articolano i cataloghi e le basi di dati verrà indicata come '`~postgres/`', ovvero la directory personale dell'utente '**postgres**', cioè il DBA (l'amministratore delle basi di dati).

Quando si installa PostgreSQL si dovrebbe avere già una directory '`~postgres/`' organizzata in modo tale da poter iniziare a creare delle basi di dati. Per questo sono necessari alcuni file, detti cataloghi, e una base di dati di partenza: '**template1**'.

324.8.1 Cataloghi del DBMS

I cataloghi di PostgreSQL sono delle tabelle del DBMS che non appartengono ad alcuna base di dati e servono per gestire il DBMS stesso. Normalmente non si dovrebbe accedere a tali tabelle direttamente, ma solo tramite script o programmi specifici. Tuttavia ci sono situazioni in cui ciò potrebbe essere necessario, tenendo conto poi che la documentazione di PostgreSQL fa spesso riferimento a queste, per cui conviene almeno saperle consultare.

Dal momento che PostgreSQL consente di accedere a delle tabelle solo dopo avere specificato la base di dati, a queste si accede attraverso '**template1**', in pratica con un comando simile a quello seguente:

```
postgres:~$ psql -d template1
```

324.8.1.1 Catalogo pg_user

Il catalogo '**pg_user**' è una vista del catalogo '**pg_shadow**', che contiene le informazioni sugli utenti di PostgreSQL. La figura 324.1 mostra un esempio di come potrebbe essere composta. La consultazione della tabella si ottiene con il comando SQL:

```
template1=> SELECT * FROM pg_user;
```

Figura 324.1. Esempio di un catalogo '**pg_user**'.

```
 usename |usesysid|usecreatedb|usetrace|usesuper|usecatupd|passwd  |valuntil
 --------|--------|-----------|--------|--------|---------|--------|----------
 postgres|     100|t          |t       |t       |t        |********|Sat Jan 31
 nobody  |      99|f          |t       |f       |t        |********|
 tizio   |    1001|t          |t       |t       |t        |********|
```

Si può osservare che l'utente '**postgres**' ha tutti gli attributi booleani attivi ('**usecreatedb**', '**usetrace**', '**usesuper**', '**usecatupd**') e questo per permettergli di compiere tutte le operazioni all'interno delle basi di dati. In particolare, l'attributo '**usecreatedb**' permette all'utente di creare una base di dati e '**usesuper**' permette di aggiungere utenti. In effetti, osservando l'esempio della figura, l'utente '**tizio**' ha praticamente gli stessi privilegi dell'amministratore '**postgres**'.

324.8.1.2 Catalogo pg_shadow

Il catalogo '**pg_shadow**' è il contenitore delle informazioni sugli utenti, a cui si accede normalmente tramite la vista '**pg_user**'. Il suo scopo è quello di conservare in un file più sicuro (perché non è accessibile agli utenti comuni) i dati delle parole d'ordine degli utenti che intendono usare le forme di autenticazione basate su queste. Per il momento, nella documentazione di PostgreSQL non viene spiegato come usarlo, né se le parole d'ordine indicate devono essere in chiaro o cifrate in qualche modo. L'esempio della figura 324.2 mostra gli stessi utenti a cui non viene abbinata alcuna parola d'ordine. La consultazione della tabella si ottiene con il comando SQL:

```
template1=> SELECT * FROM pg_shadow;
```

Figura 324.2. Esempio di un catalogo '**pg_shadow**'.

```
usename |usesysid|usecreatedb|usetrace|usesuper|usecatupd|passwd|valuntil
--------|--------|-----------|--------|--------|---------|------|----------
postgres|     100|t          |t       |t       |t        |      |Sat Jan 31
nobody  |      99|f          |t       |f       |t        |      |
tizio   |    1001|t          |t       |t       |t        |      |
```

324.8.1.3 Catalogo pg_database

Il catalogo '**pg_database**' è una tabella che contiene le informazioni sulle basi di dati esistenti. La figura 324.3 mostra un esempio di come potrebbe essere composta. La consultazione della tabella si ottiene con il comando SQL:

```
template1=> SELECT * FROM pg_database;
```

Figura 324.3. Esempio di un catalogo '**pg_database**'.

```
datname  |datdba|encoding|datpath
---------|------|--------|--------------------
template1|   100|       0|template1
pubblico |   100|       0|pubblico
prova    |   100|       0|/home/postgres/prova
prova1   |   100|       0|prova1
prova2   |  1001|       0|prova2
```

La prima colonna rappresenta il nome della base di dati, la seconda riporta il numero UID dell'utente che rappresenta il suo DBA, cioè colui che l'ha creata, la terza rappresenta il percorso in cui si trova. Per esempio, si può osservare che la base di dati '**prova2**' è stata creata dall'utente 1001, che da quanto riportato in '**pg_user**' è '**tizio**'.

La colonna che rappresenta il percorso della base di dati è più complessa da interpretare. In generale, i nomi che appaiono senza l'indicazione di un percorso si riferiscono alla directory '`~postgres/base/`', corrispondendo in pratica alla directory che contiene i file della base di dati. Per esempio, la base di dati '**prova2**' è collocata nella directory '`~postgres/base/prova2/`'. I percorsi assoluti vanno interpretati in modo speciale; in particolare, nel caso della base di dati '**prova**', la directory corrispondente è in realtà '`/home/postgres/base/prova/`' (si osservi l'inserzione di '`base/`').

In generale, è normale che tutte le basi di dati vengano create a partire da '`~postgres/base/`', pertanto non si dovrebbero vedere percorsi assoluti in questa tabella. Verrà mostrato in seguito quando può verificarsi questa condizione.

324.8.2 Copia e spostamento di una base di dati

Prima di poter pensare a copiare o a spostare una base di dati occorre avere chiaro in mente che si tratta di file «binari» (nel senso che non si tratta di file di testo), contenenti informazioni collegate l'una all'altra in qualche modo più o meno oscuro. Queste informazioni possono a volte essere espresse anche in forma numerica; in tal caso dipende dall'architettura in cui sono state create. Questo implica due cose fondamentali: lo spostamento o la copia deve essere fatto in modo che non si perdano dei pezzi per la strada (i file della stessa base di dati devono essere raccolti tutti assieme) e lo spostamento in un'altra architettura non dovrebbe essere ammissibile.

La copia di una base di dati per motivi di sicurezza è un'operazione semplice e così anche il suo ripristino. Si tratta di archiviare e poi eventualmente ripristinare tutto il contenuto della directory che la contiene. Per esempio,

```
# tar czvf base.tar.gz ~postgres/base
```

archivia nel file '`base.tar.gz`' tutte le basi di dati che si articolano a partire da '`~postgres/base/`'. Come esempio ulteriore,

```
# tar czvf pubblico.tar.gz ~postgres/base/pubblico
```

archivia nel file '`pubblico.tar.gz`' solo la base di dati '**`pubblico`**', che si trova esattamente nella directory '`~postgres/base/pubblico/`'.

Il recupero non è nulla di speciale, tranne per il fatto che si deve recuperare una base dati per intero, ovvero ciò che di solito si articola in una sottodirectory di '`~postgres/base/`'. Se di dovessero perdere informazioni sui permessi, occorre ricordare che i file devono appartenere all'utente '**`postgres`**', ovvero colui che rappresenta l'amministratore del DBMS.

Per poter spostare una base di dati nel file system occorre ricordare che l'informazione sulla sua collocazione è contenuta nel catalogo '**`pg_database`**', per cui è su questo che occorre intervenire per informare PostgreSQL della nuova posizione che gli si vuole dare. Eventualmente, c'è sempre la possibilità di eliminare la base di dati con il comando '**`dropdb`**', ricreandola nella nuova posizione, sostituendo poi tutti i file con quella vecchia. In pratica, all'interno dei file che compongono una base di dati non c'è l'informazione della loro collocazione, quindi, a parte il problema di modificare in qualche modo il catalogo '**`pg_database`**', non si dovrebbero incontrare altre difficoltà.

> Per salvare tutto il sistema di basi di dati di PostgreSQL, si può agire in modo più semplice archiviando tutta la directory '`~postgres/`', in modo ricorsivo. In questo senso, se ci sono delle basi di dati che risiedono al di fuori della gerarchia '`~postgres/`', le cose si complicano, così si spiega il motivo dell'organizzazione standard di PostgreSQL che prevede la loro collocazione al di sotto della gerarchia '`~postgres/base/`'. Nel caso non fosse ancora chiaro, è bene ribadire che salvando anche i file che risiedono esattamente nella directory '`~postgres/`', si evita di dover ricreare le basi di dati prima del loro recupero, ovvero si evita di dover intervenire manualmente nei cataloghi per dichiararne la presenza.

324.8.3 Creazione di una base di dati in una collocazione diversa dalla solita

Il comando '**createdb**', se non viene specificato diversamente, crea la base di dati in una sottodirectory a partire da '`~postgres/base/`'. Se si vuole definire una nuova posizione basta usare l'opzione '**-D**', seguita dalla directory che deve essere presa in considerazione al posto di '`~postgres/`' (ovvero di '**PGDATA**' come si legge nella documentazione di PostgreSQL).

Perché la cosa funzioni, occorre che la directory ricevente sia pronta. Per esempio, volendo creare la base di dati '**mia**' a partire da '`/home/postgresql/`', sapendo che poi in pratica la sottodirectory '`mia/`' viene collocata su '`/home/postgresql/base/`', occorre predisporre tutto questo.

```
# mkdir /home/postgresql

# mkdir /home/postgresql/base

# chown -R postgres. /home/postgresql
```

La preparazione delle directory può essere fatta con l'aiuto di '**initlocation**', ma questo comando non fa niente di particolare in più. Per completare l'esempio, viene mostrato il comando con cui si crea la base di dati '**mia**', utilizzando come riferimento la directory '`/home/postgresql`'.

```
postgres:~$ createdb -D /home/postgresql mia
```

Per concludere, se si osserva il catalogo '**pg_database**', si noterà che il percorso indicato della base di dati appena creata è '`/home/postgresql/mia/`', mentre invece la directory vera e propria è '`/home/postgresql/base/mia/`'.

324.8.4 Copia e spostamento di una base di dati, in modo indipendente dalla piattaforma

Dopo aver visto in che modo è possibile copiare e archiviare una base di dati, rimanendo sulla stessa piattaforma, ma soprattutto, rimanendo nell'ambito della stessa versione di PostgreSQL, è necessario vedere in che modo si può risolvere il problema quando la piattaforma cambia, o quando cambia la versione di PostgreSQL.

Ricapitolando, quindi, i problemi sono due: la piattaforma e la versione di PostgreSQL. In linea di principio, non è possibile copiare una base di dati realizzata su GNU/Linux in una macchina i386 per portarla in un'altra macchina con architettura differente, anche se con lo stesso sistema operativo; nemmeno si può trasportare una base di dati, così come si trova, da un sistema operativo a un altro. Inoltre, PostgreSQL non è in grado di leggere, o di utilizzare in alcun modo, le basi di dati realizzate con altre versioni dello stesso.

Attualmente, l'unico modo per raggirare l'ostacolo è lo scarico dei dati (*dump*) in uno script che successivamente può essere dato in pasto a '**psql**', per ricreare le basi di dati come erano in origine. Naturalmente, questa tecnica non è perfetta e funziona correttamente solo quando le basi di dati non contengono relazioni con tuple eccessivamente grandi.

> Questo problema deve essere preso in considerazione già nel momento della progettazione di una base di dati, avendo cura di verificare, sperimentandolo, che il procedimento di scarico e recupero dei dati possa funzionare.

Lo scarico di una base di dati si ottiene attraverso il programma '**pg_dump**', che è parte integrante della distribuzione di PostgreSQL.

```
pg_dump [opzioni] base_di_dati
```

Se non si indicano delle opzioni e ci si limita a specificare la base di dati su cui intervenire, si ottiene il risultato attraverso lo standard output, composto in pratica dai comandi necessari a '**psql**' per ricostruire le relazioni che compongono la base di dati (la base di dati stessa deve essere ricreata manualmente). Tanto per chiarire subito il senso della cosa, se si utilizza '**pg_dump**' nel modo seguente,

```
$ pg_dump mio_db > mio_db.dump
```

si ottiene il file di testo '`mio_db.dump`'. Questo file va verificato, ricercando la presenza eventuale di segnalazioni di errore che vengono generate in presenza di dati che non possono essere riprodotti fedelmente; eventualmente, il file può anche essere modificato se si conosce la sintassi dei comandi che vengono inseriti in questo script. Per fare in modo che le relazioni della base di dati vengano ricreate e caricate, si può utilizzare '**psql**' nel modo seguente:

```
$ psql -e mio_db < mio_db.dump
```

Alcune opzioni

`-d`

In condizioni normali, '**pg_dump**' salva i dati delle relazioni (le tabella secondo l'SQL) in una forma compatibile con il comando '**COPY**', che però non è compatibile con lo standard SQL. Con l'opzione '**-d**', utilizza il comando '**INSERT**' tradizionale.

`-D`

Come con l'opzione '**-d**', con l'aggiunta dell'indicazione degli attributi (le colonne secondo l'SQL) in cui vanno inseriti i dati. In pratica, questa opzione permette di generare uno script più preciso e dettagliato.

`-f` *file*

Permette di definire un file diverso dallo standard output, che si vuole generare con il risultato dell'elaborazione di '**pg_dump**'.

`-h` *host*

Permette di specificare il nodo a cui connettersi per l'interrogazione del servente PostgreSQL. In pratica, se l'accesso è consentito, è possibile scaricare una base di dati gestita presso un nodo remoto.

`-p` *porta*

Nel caso in cui '**postmaster**' sia in ascolto su una porta TCP diversa dal numero 5432 (corrispondente al valore predefinito), si può specificare con questa opzione il numero corretto da utilizzare.

`-s`

Scarica soltanto la struttura delle relazioni, senza occuparsi del loro contenuto. In pratica, serve per poter riprodurre vuote le tabelle SQL.

`-t` *nome_tabella*

Utilizzando questa opzione, indicando il nome di una tabella SQL, si ottiene lo scarico di quell'unica tabella.

`-u`

Fa in modo che '**psql**' richieda il nominativo-utente e la parola d'ordine all'utente, prima di tentare la connessione. L'uso di questa opzione è indispensabile quando il servente impone una forma di autenticazione definita attraverso la parola chiave '**password**'.

`-z`

Include le informazioni sui permessi e la proprietà delle tabelle ('**GRANT**'/'**REVOKE**').

324.8.5 Copia, spostamento e aggiornamento di tutte le basi di dati, in modo indipendente dalla piattaforma

Per copiare o trasferire tutte le basi di dati del sistema di PostgreSQL, si può utilizzare '**pg_dumpall**', che in pratica è uno script che si avvale di '**pg_dump**' per compiere il suo lavoro.

`pg_dumpall` [*opzioni*]

'**pg_dumpall**' provvede a scaricare tutte le basi di dati, assieme alle informazioni necessarie per ricreare il catalogo '**pg_shadow**' (la vista '**pg_user**' si ottiene di conseguenza). Come si può intuire, si deve utilizzare '**pg_dumpall**' con i privilegi dell'utente '**postgres**'.

Gli argomenti della riga di comando di '**pg_dumpall**', vengono passati tali e quali a '**pg_dump**', quando questo viene utilizzato all'interno dello script per lo scarico di ogni singola base di dati.

```
postgres$ pg_dumpall > basi_dati.dump
```

L'esempio mostra il modo più semplice di utilizzare '**pg_dumpall**' per scaricare tutte le basi di dati in un file unico. In questo caso, si ottiene il file di testo '`basi_dati.dump`'. Questo file va verificato alla ricerca di segnalazioni di errore che potrebbero essere generate in presenza di dati che non possono essere riprodotti fedelmente; eventualmente, può essere modificato se si conosce la sintassi dei comandi che vengono inseriti in questo script.

Il recupero dell'insieme completo delle basi di dati avviene normalmente in un'ambiente PostgreSQL, in cui il sistema delle basi di dati sia stato predisposto, ma non sia stata creata alcuna base di dati (a parte '**template1**' la cui presenza è obbligatoria). Come si può intuire, il comando necessario per ricaricare le basi di dati, assieme alle informazioni sugli utenti (il catalogo '**pg_shadow**'), è quello seguente:

```
postgres$ psql -e template1 < basi_dati.dump
```

La situazione tipica in cui è necessario utilizzare '**pg_dumpall**' per scaricare tutto il sistema delle basi di dati, è quella del momento in cui ci si accinge ad aggiornare la versione di PostgreSQL. In breve, in quella occasione, si devono eseguire i passaggi seguenti:

1. con la versione vecchia di PostgreSQL, si deve utilizzare '**pg_dumpall**' in modo da scaricare tutto il sistema delle basi di dati in un solo file di testo;
2. si aggiorna PostgreSQL;
3. si elimina il contenuto della directory '`~postgres/`', ovvero quella che altrimenti viene definita '**PGDATA**' (prima conviene forse fare una copia di sicurezza);
4. si ricrea il sistema delle basi di dati, vuoto, attraverso '**initdb**';
5. si ricaricano le basi di dati precedenti, assieme alle informazioni sugli utenti, attraverso '**psql**', utilizzando il file generato in precedenza attraverso '**pg_dumpall**'.

Quello che manca, di solito si tratta del file '`~postgres/pg_hda.conf`' per la configurazione dei sistemi di accesso e autenticazione, deve essere ripristinato manualmente.

324.9 Riferimenti

- *PostgreSQL*

  <http://www.postgresql.org/>

- Bruce Momjian, *PostgreSQL: introduction and concepts*

  <http://www.postgresql.org/docs/awbook.html>

- Al Dev (Alavoor Vasudevan), *PostgreSQL HOWTO*

  <http://www.linux.org/docs/ldp/howto/HOWTO-INDEX/howtos.html>

PostgreSQL: il linguaggio

PostgreSQL è un ORDBMS, ovvero un *Object-relational DBMS,* cioè un DBMS relazionale a oggetti. La sua documentazione utilizza terminologie differenti, a seconda delle preferenze dei rispettivi autori. In generale si possono distinguere tre modalità, riferite a tre punti di vista: la programmazione a oggetti, la teoria generale sui DBMS e il linguaggio SQL. Le equivalenze dei termini sono riassunte dall'elenco seguente:

- classi, istanze, attributi e tipi di dati contenibili negli attributi;
- relazioni, tuple, attributi e domini;
- tabelle, righe, colonne e tipi di dati contenibili nelle colonne.

In questo capitolo si intende usare la terminologia tradizionale dei DBMS relazioni, corrispondente a quella delle prime versioni del linguaggio SQL: tabelle, righe, colonne,... Nello stesso modo, la sintassi delle istruzioni (interrogazioni) SQL che vengono mostrate è limitata alle funzionalità più semplici, sempre compatibilmente con le possibilità di PostgreSQL. Per una visione più estesa delle funzionalità SQL di PostgreSQL conviene consultare la sua documentazione, a cominciare da *sql(1)* per finire con il manuale dell'utente che contiene diversi esempi molto utili.

325.1 Prima di iniziare

Per fare pratica con il linguaggio SQL, il modo migliore è quello di utilizzare il programma '**`psql`**' con il quale si possono eseguire interrogazioni interattive con il servente. Quello che conta è tenere a mente che per poterlo utilizzare occorre avere già creato una base di dati (vuota), in cui verranno inserite delle nuove tabelle, con le quali si eseguiranno altre operazioni.

Attraverso le istruzioni SQL si fa riferimento sempre a un'unica base di dati: quella a cui ci si collega quando si avvia '**`psql`**'.

> Utilizzando '**`psql`**', le istruzioni devono essere terminate con il punto e virgola ('**`;`**'), oppure dal comando interno '**`\g`**' (*go*).

325.2 Tipi di dati e rappresentazione

I tipi di dati gestibili sono un punto delicato della compatibilità tra un DBMS e lo standard SQL. Vale la pena di riepilogare i tipi più comuni, compatibili con lo standard SQL, che possono essere trovati nella tabella 325.1. Si deve tenere presente che SQL utilizza diversi modi possibili per definire lo stesso tipo di dati; per esempio il tipo '**`CHAR`**' può essere indicato anche come '**`CHARACTER`**', così pure '**`VARCHAR`**' può essere espresso come '**`CHAR VARYING`**' o '**`CHARACTER VARYING`**'. Quando PostgreSQL ammette l'utilizzo di una forma, riconosce poi anche le altre.

Tabella 325.1. Elenco dei tipi di dati standard utilizzabili con PostgreSQL, espressi nella loro forma compatta.

Tipo	Standard	Descrizione
CHAR	SQL92	Un carattere singolo.
CHAR(*n*)	SQL92	Una stringa di lunghezza fissa, di *n* caratteri, completata da spazi.
VARCHAR(*n*)	SQL92	Una stringa di lunghezza variabile con un massimo di *n* caratteri.
INTEGER	SQL92	Intero (al massimo nove cifre numeriche).
SMALLINT	SQL92	Intero più piccolo di '**INTEGER**'.
FLOAT	SQL92	Numero a virgola mobile.
FLOAT(*n*)	SQL92	Numero a virgola mobile lungo *n* bit.
REAL	SQL92	Numero a virgola mobile (teoricamente più preciso di '**FLOAT**').
DOUBLE PRECISION	SQL92	Numero a virgola mobile (più o meno equivalente a '**REAL**').
DATE	SQL92	Data, di solito nella forma '*mm*`/`*gg*`/`*aaaa*''.
TIME	SQL92	Orario, nella forma '*hh*`:`*mm*`:`*ss*', oppure solo '*hh*`:`*mm*'.
TIMESTAMP	SQL92	Informazione completa data-orario.
INTERVAL	SQL92	Intervallo di tempo.
BOOLEAN	SQL3	Valore logico booleano.

Oltre ai tipi di dati gestibili, è necessario conoscere il modo di rappresentarli in forma costante. In particolare, è bene osservare che PostgreSQL ammette solo l'uso degli apici singoli come delimitatori. La tabella 325.2 mostra alcuni esempi.

Tabella 325.2. Esempi di rappresentazione dei valori costanti.

Tipo	Esempi
CHAR	'**'a'**', '**'A'**', '**'b'**', '**'1'**'.
CHAR(*n*)	'**'a'**', '**'ciao'**', '**'Ciao'**', '**'123/der:876'**'.
VARCHAR(*n*)	'**'a'**', '**'ciao'**', '**'Ciao'**', '**'123/der:876'**'.
INTEGER	'**1**', '**123**', '**-987**'.
SMALLINT	Come '**INTEGER**'.
FLOAT	'**123.45**', '**-45.3**', '**123.45e+10**', '**123.45e-10**'.
FLOAT(*n*)	Come '**FLOAT**'.
REAL	Come '**FLOAT**'.
DOUBLE PRECISION	Come '**FLOAT**'.
DATE	'**'31.12.1999'**', '**'12/31/1999'**', '**'1999-12-31'**'.
TIME	'**'15:55:27'**', '**'15:59'**'.
TIMESTAMP	'**'1999-12-31 15:55:27'**', '**'1999-12-31 15:55:27+1'**'.
INTERVAL	'**INTERVAL '15:55:27'**', '**INTERVAL '15 HOUR 59 MINUTE'**', '**INTERVAL '- 15 HOUR'**'.
BOOLEAN	'**1**', '**'y'**', '**'yes'**', '**'t'**', '**'true'**'; '**0**', '**'n'**', '**'no'**', '**'f'**', '**'false'**'.

In particolare, le costanti stringa possono contenere delle sequenze di escape, rappresentate da una barra obliqua inversa seguita da un simbolo. La tabella 325.3 mostra le sequenze di escape tipiche.

Tabella 325.3. Sequenze di escape utilizzabili all'interno delle stringhe di caratteri costanti.

Escape	Significato
\n	*newline*
\r	*return*
\b	*backspace*
\'	'
\"	"
\\	\
\%	%
_	_

325.3 Funzioni

PostgreSQL, come altri DBMS SQL, offre una serie di funzioni che fanno parte dello standard SQL, assieme ad altre non standard che però sono ampiamente diffuse e di grande utilità. Le tabelle 325.4 e 325.5 ne riportano alcune.

Tabella 325.4. Funzioni SQL riconosciute da PostgreSQL.

Funzione	Descrizione
POSITION(*stringa_1* IN *stringa_2*)	Posizione di *stringa_1* in *stringa_2*.
SUBSTRING(*stringa* [FROM *n*] [FOR *m*])	Sottostringa da *n* per *m* caratteri.
TRIM([LEADING|TRAILING|BOTH] ['*x*'] FROM [*stringa*])	Ripulisce all'inizio e alla fine del testo.

Tabella 325.5. Alcune funzioni riconosciute dal linguaggio di PostgreSQL.

Funzione	Descrizione
UPPER(*stringa*)	Converte la stringa in caratteri maiuscoli.
LOWER(*stringa*)	Converte la stringa in caratteri minuscoli.
INITCAP(*stringa*)	Converte la stringa in modo che le parole inizino con la maiuscola.
SUBSTR(*stringa*,*n*,*m*)	Estrae la stringa che inizia dalla posizione *n*, lunga *m* caratteri.
LTRIM(*stringa*, '*x*')	Ripulisce la stringa a sinistra (*Left trim*).
RTRIM(*stringa*, '*x*')	Ripulisce la stringa a destra (*Right trim*).

Esempi

```
SELECT POSITION( 'o' IN 'Topo' )
```

Restituisce il valore due.

```
SELECT POSITION( 'ino' IN Cognome ) FROM Indirizzi
```

Restituisce un elenco delle posizioni in cui si trova la stringa '**ino**' all'interno della colonna '**Cognome**', per tutte le righe della tabella '**Indirizzi**'.

```
SELECT SUBSTRING( 'Daniele' FROM 3 FOR 2 )
```

Restituisce la stringa '**ni**'.

```
SELECT TRIM( LEADING '*' FROM '*****Ciao****' )
```

Restituisce la stringa '**Ciao****'.

```
SELECT TRIM( TRAILING '*' FROM '*****Ciao****' )
```

Restituisce la stringa '*****Ciao**'.

```
SELECT TRIM( BOTH '*' FROM '*****Ciao****' )
```

Restituisce la stringa '**Ciao**'.

```
SELECT TRIM( BOTH ' ' FROM '    Ciao    ' )
```

Restituisce la stringa '**Ciao**'.

```
SELECT TRIM( '    Ciao    ' )
```

Esattamente come nell'esempio precedente, dal momento che lo spazio normale è il carattere predefinito e che la parola chiave '**BOTH**' è anche predefinita.

```
SELECT LTRIM( '*****Ciao****', '*' )
```

Restituisce la stringa '**Ciao****'.

```
SELECT RTRIM( '*****Ciao****', '*' )
```

Restituisce la stringa '*****Ciao**'.

325.4 Esempi comuni

Nelle sezioni seguenti vengono mostrati alcuni esempi comuni di utilizzo del linguaggio SQL, limitato alle possibilità di PostgreSQL. La sintassi non viene descritta, salvo quando la differenza tra quella standard e quella di PostgreSQL è importante.

Negli esempi si fa riferimento frequentemente a una tabella di indirizzi, il cui contenuto è visibile nella figura 325.1.

Figura 325.1. La tabella '**Indirizzi(Codice,Cognome,Nome,Indirizzo,Telefono)**' usata in molti esempi del capitolo.

```
.=============================================================================.
|Indirizzi                                                                    |
|-----------------------------------------------------------------------------|
|Codice|Cognome        |Nome           |Indirizzo      |Telefono      |
|------|---------------|---------------|---------------|--------------|
|     1|Pallino        |Pinco          |Via Biglie 1   |0222,222222   |
|     2|Tizi           |Tizio          |Via Tazi 5     |0555,555555   |
|     3|Cai            |Caio           |Via Caini 1    |0888,888888   |
|     4|Semproni       |Sempronio      |Via Sempi 7    |0999,999999   |
'=============================================================================='
```

325.4.1 Creazione di una tabella

La tabella di esempio mostrata nella figura 325.1, potrebbe essere creata nel modo seguente:

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40),
        Nome            char(40),
        Indirizzo       varchar(60),
        Telefono        varchar(40)
    );
```

Quando si inseriscono i valori per una riga, può capitare che venga omesso l'inserimento di alcune colonne. In questi casi, il campo corrispondente riceve il valore '**NULL**', cioè un valore indefinito, oppure il valore predefinito attraverso quanto specificato con l'espressione che segue la parola chiave '**DEFAULT**'.

In alcuni casi non è possibile definire un valore predefinito e nemmeno è accettabile che un dato resti indefinito. In tali situazioni si può aggiungere '**NOT NULL**', dopo la definizione del tipo.

325.4.2 Modifica della tabella

Per il momento, le funzionalità di modifica della struttura di una tabella sono limitate alla sola aggiunta di colonne, come nell'esempio seguente dove viene aggiunta una colonna per l'indicazione del comune di residenza alla tabella già vista in precedenza.

```
ALTER TABLE Indirizzi ADD COLUMN Comune char(30);
```

È bene osservare che non sempre si ottiene il risultato desiderato.

325.4.3 Inserimento dati in una tabella

L'esempio seguente mostra l'inserimento dell'indirizzo dell'impiegato «Pinco Pallino».

```
INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    );
```

In questo caso, si presuppone che i valori inseriti seguano la sequenza delle colonne, così come è stata creata la tabella in origine. Se si vuole indicare un comando più leggibile, occorre aggiungere l'indicazione della sequenza delle colonne da compilare, come nell'esempio seguente:

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
        Indirizzo,
        Telefono
    )
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    );
```

In questo stesso modo, si può evitare di compilare il contenuto di una colonna particolare, indicando espressamente solo le colonne che si vogliono fornire; le altre colonne riceveranno il valore predefinito o '**NULL**' in mancanza d'altro. Nell'esempio seguente viene indicato solo il codice e il nominativo.

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
    )
    VALUES (
        01,
        'Pallino'
        'Pinco',
    );
```

325.4.4 Eliminazione di una tabella

Una tabella può essere eliminata completamente attraverso l'istruzione '**DROP**'. L'esempio seguente elimina la tabella degli indirizzi degli esempi precedenti.

```
DROP TABLE Indirizzi;
```

325.4.5 Interrogazioni semplici

L'esempio seguente emette tutto il contenuto della tabella degli indirizzi già vista negli esempi precedenti.

```
SELECT * FROM Indirizzi;
```

Seguendo l'esempio fatto in precedenza si dovrebbe ottenere l'elenco riportato sotto, equivalente a tutto il contenuto della tabella.

```
codice  cognome   nome       indirizzo     telefono

     1  Pallino   Pinco      Via Biglie 1  0222,222222
     2  Tizi      Tizio      Via Tazi 5    0555,555555
     3  Cai       Caio       Via Caini 1   0888,888888
     4  Semproni  Sempronio  Via Sempi 7   0999,999999
```

Per ottenere un elenco ordinato in base al cognome e al nome (in caso di ambiguità), lo stesso comando si completa nel modo seguente:

```
SELECT * FROM Indirizzi ORDER BY Cognome, Nome;
```

```
codice  cognome   nome       indirizzo     telefono

     3  Cai       Caio       Via Caini 1   0888,888888
     1  Pallino   Pinco      Via Biglie 1  0222,222222
     4  Semproni  Sempronio  Via Sempi 7   0999,999999
     2  Tizi      Tizio      Via Tazi 5    0555,555555
```

La selezione delle colonne permette di ottenere un risultato con le sole colonne desiderate, permettendo anche di cambiarne l'intestazione. L'esempio seguente permette di mostrare solo i nominativi e il telefono, cambiando un po' le intestazioni.

```
SELECT Cognome as cognomi, Nome as nomi, Telefono as numeri_telefonici
    FROM Indirizzi;
```

Quello che si ottiene è simile all'elenco seguente:

```
cognomi   nomi       numeri_telefonici

Pallino   Pinco      0222,222222
Tizi      Tizio      0555,555555
Cai       Caio       0888,888888
Semproni  Sempronio  0999,999999
```

La selezione delle righe può essere fatta attraverso la condizione che segue la parola chiave '**WHERE**'. Nell'esempio seguente vengono selezionate le righe in cui l'iniziale dei cognomi è compresa tra '**N**' e '**T**'.

```
SELECT * FROM Indirizzi WHERE Cognome >= 'N' AND Cognome <= 'T';
```

Dall'elenco che si ottiene, si osserva che '**Caio**' è stato escluso.

```
codice  cognome   nome       indirizzo     telefono

     1  Pallino   Pinco      Via Biglie 1  0222,222222
     2  Tizi      Tizio      Via Tazi 5    0555,555555
     4  Semproni  Sempronio  Via Sempi 7   0999,999999
```

Come si vedrà meglio in seguito, per evitare ambiguità possono essere indicati i nomi delle colonne prefissati dal nome della tabella a cui appartengono, separando le due parti con l'operatore punto ('**.**'). L'esempio seguente è già stato mostrato in precedenza, ma serve a chiarire questo modo di identificazione delle colonne.

```
SELECT Indirizzi.Cognome, Indirizzi.Nome, Indirizzi.Telefono
    FROM Indirizzi;
```

```
cognome   nome       telefono

Pallino   Pinco      0222,222222
Tizi      Tizio      0555,555555
Cai       Caio       0888,888888
Semproni  Sempronio  0999,999999
```

325.4.6 Interrogazioni simultanee di più tabelle

Se dopo la parola chiave '**FROM**' si indicano più tabelle (ciò vale anche se si indica più volte la stessa tabella), si intende fare riferimento a una tabella generata dal «prodotto» di queste. Si immagini di abbinare alla tabella '**Indirizzi**' la tabella '**Presenze**' contenente i dati visibili nella figura 325.2.

Figura 325.2. La tabella '**Presenze(Codice,Giorno,Ingresso,Uscita)**'.

```
.================================.
|Presenze                        |
|--------------------------------|
|Codice|Giorno    |Ingresso|Uscita|
|------|----------|--------|------|
|     1|01/01/1999| 07:30  |13:30 |
|     2|01/01/1999| 07:35  |13:37 |
|     3|01/01/1999| 07:45  |14:00 |
|     4|01/01/1999| 08:30  |16:30 |
|     1|01/02/1999| 07:35  |13:38 |
|     2|01/02/1999| 08:35  |14:37 |
|     4|01/02/1999| 07:40  |13:30 |
'================================'
```

Come si può intendere, la prima colonna, '**Codice**', serve a identificare la persona per la quale è stata fatta l'annotazione dell'ingresso e dell'uscita. Tale codice viene interpretato in base al contenuto della tabella '**Indirizzi**'. Si immagini di volere ottenere un elenco contenente tutti gli ingressi e le uscite, indicando chiaramente il cognome e il nome della persona a cui si riferiscono.

```
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

Ecco quello che si dovrebbe ottenere.

```
giorno      ingresso  uscita    cognome   nome

01-01-1999  07:30:00  13:30:00  Pallino   Pinco
01-01-1999  07:35:00  13:37:00  Tizi      Tizio
01-01-1999  07:45:00  14:00:00  Cai       Caio
01-01-1999  08:30:00  16:30:00  Semproni  Sempronio
01-02-1999  07:35:00  13:38:00  Pallino   Pinco
01-02-1999  08:35:00  14:37:00  Tizio     Tizi
01-02-1999  07:40:00  13:30:00  Semproni  Sempronio
```

325.4.7 Alias

Una stessa tabella può essere presa in considerazione come se si trattasse di due o più tabelle differenti. Per distinguere tra questi punti di vista diversi, si devono usare degli alias, che sono in pratica dei nomi alternativi. Gli alias si possono usare anche solo per questioni di leggibilità. L'esempio seguente è la semplice ripetizione di quello mostrato nella sezione precedente, con l'aggiunta però della definizione degli alias '**Pre**' e '**Nom**'.

```
SELECT
    Pre.Giorno,
    Pre.Ingresso,
    Pre.Uscita,
    Nom.Cognome,
    Nom.Nome
    FROM Presenze AS Pre, Indirizzi AS Nom
    WHERE Pre.Codice = Nom.Codice;
```

325.4.8 Viste

Attraverso una vista, è possibile definire una tabella virtuale. PostgreSQL, allo stato attuale, consente di utilizzare le viste in sola lettura.

```
CREATE VIEW Presenze_dettagliate AS
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

L'esempio mostra la creazione della vista '**`Presenze_dettagliate`**', ottenuta dalle tabelle '**`Presenze`**' e '**`Indirizzi`**'. In pratica, questa vista permette di interrogare direttamente la tabella virtuale '**`Presenze_dettagliate`**', invece di utilizzare ogni volta un comando '**`SELECT`**' molto complesso, per ottenere lo stesso risultato.

325.4.9 Aggiornamento delle righe

La modifica di righe già esistenti avviene attraverso l'istruzione '**`UPDATE`**', la cui efficacia viene controllata dalla condizione posta dopo la parola chiave '**`WHERE`**'. Se tale condizione manca, l'effetto delle modifiche si riflette su tutte le righe della tabella.

L'esempio seguente, aggiunge una colonna alla tabella degli indirizzi, per contenere il nome del comune di residenza degli impiegati; successivamente viene inserito il nome del comune '**`Sferopoli`**' in base al prefisso telefonico.

```
ALTER TABLE Indirizzi ADD COLUMN Comune char(30);
```

```
UPDATE Indirizzi
    SET Comune='Sferopoli'
    WHERE Telefono >= '022' AND Telefono < '023';
```

In pratica, viene aggiornata solo la riga dell'impiegato '**`Pinco Pallino`**'.

325.4.10 Cancellazione delle righe

L'esempio seguente elimina dalla tabella delle presenze le righe riferite alle registrazioni del giorno 01/01/1999 e le eventuali antecedenti.

```
DELETE FROM Presenze WHERE Giorno <= '01/01/1999';
```

325.4.11 Creazione di una nuova tabella a partire da altre

L'esempio seguente crea la tabella '**mia_prova**' come risultato della fusione della tabella degli indirizzi e delle presenze, come già mostrato in un esempio precedente.

```
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    INTO TABLE mia_prova
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

325.4.12 Inserimento in una tabella esistente

L'esempio seguente aggiunge alla tabella dello storico delle presenze le registrazioni vecchie che poi vengono cancellate.

```
INSERT INTO PresenzeStorico (
        PresenzeStorico.Codice,
        PresenzeStorico.Giorno,
        PresenzeStorico.Ingresso,
        PresenzeStorico.Uscita
    )
    SELECT
        Presenze.Codice,
        Presenze.Giorno,
        Presenze.Ingresso,
        Presenze.Uscita
        FROM Presenze
        WHERE Presenze.Giorno <= '1999/01/01';

DELETE FROM Presenze WHERE Giorno <= '1999/01/01';
```

325.4.13 Controllare gli accessi a una tabella

Quando si creano delle tabelle in una base di dati, tutti gli altri utenti che sono stati registrati nel sistema del DBMS, possono accedervi e fare le modifiche che vogliono. Per controllare questi accessi, l'utente proprietario delle tabelle (cioè colui che le ha create), può usare le istruzioni '**GRANT**' e '**REVOKE**'. La prima permette a un gruppo di utenti di eseguire operazioni determinate, la seconda toglie dei privilegi.

```
GRANT {ALL | SELECT | INSERT | UPDATE | DELETE | RULE} [,...]
    ON tabella [,...]
    TO {PUBLIC | GROUP gruppo | utente}
```

```
REVOKE {ALL | SELECT | INSERT | UPDATE | DELETE | RULE} [,...]
    ON tabella [,...]
    FROM {PUBLIC | GROUP gruppo | utente}
```

La sintassi delle due istruzioni è simile, basta fare attenzione a cambiare la parola chiave '**TO**' con '**FROM**'. I gruppi e gli utenti sono nomi che fanno riferimento a quanto registrato all'interno del DBMS; solo che attualmente potrebbe non essere possibile la gestione dei gruppi.

L'esempio seguente toglie a tutti gli utenti ('**PUBLIC**') tutti i privilegi sulle tabelle delle presenze e degli indirizzi; successivamente vengono ripristinati tutti i privilegi solo per l'utente '**daniele**'.

```
REVOKE ALL
    ON Presenze, Indirizzi
    FROM PUBLIC;

GRANT ALL
    ON Presenze, Indirizzi
    TO daniele;
```

325.5 Controllo delle transazioni

PostgreSQL ha una gestione delle transazioni leggermente diversa da quanto stabilito dall'SQL. Per la precisione, occorre dichiarare esplicitamente l'inizio di una transazione con l'istruzione '**BEGIN**'.

```
BEGIN [WORK]
```

L'esempio seguente mostra il caso in cui si voglia isolare l'inserimento di una riga nella tabella '**Indirizzi**' all'interno di una transazione, che alla fine viene confermata regolarmente.

```
BEGIN;

INSERT INTO Indirizzi
    VALUES (
        05,
        'De Pippo',
        'Pippo',
        'Via Pappo, 5',
        '0333,3333333'
    );

COMMIT;
```

Nell'esempio seguente, si rinuncia all'inserimento della riga con l'istruzione '**ROLLBACK**' finale.

```
BEGIN;

INSERT INTO Indirizzi
    VALUES (
        05,
        'De Pippo',
        'Pippo',
        'Via Pappo, 5',
        '0333,3333333'
    );

ROLLBACK;
```

325.6 Cursori

La gestione dei cursori da parte di PostgreSQL è limitata rispetto all'SQL92. In particolare, non è disponibile lo spostamento assoluto del cursore, inoltre non è possibile assegnare i dati a delle variabili.

La dichiarazione di un cursore si ottiene nel modo solito, con la differenza che questa deve avvenire esplicitamente in una transazione. In particolare, con PostgreSQL, il cursore viene aperto automaticamente nel momento della dichiarazione, per cui l'istruzione '**OPEN**' non è disponibile.

```
BEGIN;

DECLARE Mio_cursore INSENSITIVE CURSOR FOR
    SELECT * FROM Indirizzi ORDER BY Cognome, Nome;
```

```
-- L'apertura del cursore non esiste in PostgreSQL
-- OPEN Mio_cursore;
...
```

L'esempio mostra la dichiarazione dell'inizio di una transazione, assieme alla dichiarazione del cursore '**Mio_cursore**', per selezionare tutta la tabella '**Indirizzi**' in modo ordinato per '**Cognome**'. Si osservi che per PostgreSQL la selezione che si ingloba nella gestione di un cursore non può aggiornarsi automaticamente se i dati originali cambiano, per cui è come se fosse sempre definita la parola chiave '**INSENSITIVE**'.

```
...
FETCH NEXT FROM Mio_cursore;
...
COMMIT;
```

L'esempio mostra l'uso tipico dell'istruzione '**FETCH**', in cui si preleva la prossima riga rispetto alla posizione corrente del cursore e più avanti si conclude la transazione con un '**COMMIT**'. L'esempio seguente è identico, con la differenza che si indica espressamente il passo.

```
...
FETCH RELATIVE 1 FROM Mio_cursore;
...
COMMIT;
```

Un cursore dovrebbe essere chiuso attraverso una richiesta esplicita, con l'istruzione '**CLOSE**', ma la chiusura della transazione chiude implicitamente il cursore, se questo dovesse essere rimasto aperto. L'esempio seguente riepiloga quanto visto sopra, completato dell'istruzione '**CLOSE**'.

```
BEGIN;

DECLARE Mio_cursore INSENSITIVE CURSOR FOR
    SELECT * FROM Indirizzi ORDER BY Cognome, Nome;

-- L'apertura del cursore non esiste in PostgreSQL
-- OPEN Mio_cursore;

FETCH NEXT FROM Mio_cursore;

CLOSE Mio_cursore;

COMMIT;
```

325.7 Particolarità di PostgreSQL

Ogni DBMS, per quanto compatibile con gli standard, può avere la necessità di introdurre delle estensioni al linguaggio di gestione per permettere l'accesso a funzionalità speciali che dipendono dalle sue caratteristiche particolari. In questo capitolo si è voluto porre l'accento su ciò che è il più vicino possibile all'SQL, trascurando quasi tutto il resto. In queste sezioni si descrivono alcune istruzioni particolari che si ritengono importanti da un punto di vista operativo, benché siano estranee all'SQL.

325.7.1 Importazione ed esportazione dei dati

PostgreSQL fornisce un'istruzione speciale per permettere l'importazione e l'esportazione dei dati da e verso un file di testo normale. Si tratta di '**COPY**' la cui sintassi semplificata è quella seguente:

```
COPY tabella FROM { 'file' | STDIN }
    [ USING DELIMITERS 'delimitatore' ]
```

```
COPY tabella TO { 'file' | STDOUT }
    [ USING DELIMITERS 'delimitatore' ]
```

Nella prima delle due forme, si importano i dati da un file o dallo standard input; nel secondo si esportano verso un file o verso lo standard output.

Ogni riga del file di testo corrisponde a una riga della tabella; gli attributi sono separati da un carattere di delimitazione, che in mancanza della definizione tramite la clausola '**USING DELIMITERS**' è un carattere di tabulazione. In ogni caso, anche se si specifica tale clausola, può trattarsi solo di un carattere. In pratica, ogni riga è organizzata secondo lo schema seguente:

```
attributo_1 x attributo_2 x ... x attributo_N
```

Nello schema, *x* rappresenta il carattere di delimitazione, che, come si può vedere, non viene inserito all'inizio e alla fine.

Quando l'istruzione '**COPY**' viene usata per importare dati dallo standard input, è necessario che dopo l'ultima riga che contiene attributi da inserire nella tabella, sia presente una sequenza di escape speciale: una barra obliqua inversa seguita da un punto ('**\.**'). Il file ottenuto quando si esporta verso lo standard output contiene questo simbolo di conclusione.

Il file di testo in questione può contenere anche altre sequenze di escape, che si trovano descritte nella tabella 325.6.

Tabella 325.6. Sequenze di escape nei file di testo generati e utilizzati da '**COPY**'.

Escape	Descrizione
\\	Una barra obliqua inversa.
\.	Simbolo di conclusione del file.
\N	'**NULL**'.
delimitatore	Protegge il simbolo che viene già utilizzato come delimitatore.
\<*LF*>	Tratta <*LF*> in modo letterale.

È importante fare mente locale al fatto che l'istruzione viene eseguita dal servente. Ciò significa che i file di testo, quando non si tratta di standard input o di standard output, sono creati o cercati secondo il file system che questo servente si trova ad avere sotto di sé.

```
COPY Indirizzi TO STDOUT;
```

L'esempio mostra l'istruzione necessaria a emettere attraverso lo standard output del programma cliente ('**psql**') la trasformazione in testo del contenuto della tabella '**Indirizzi**',

```
COPY Indirizzi TO '/tmp/prova' USING DELIMITERS '|';
```

In questo caso, si genera il file '`/tmp/prova`' nel file system dell'elaboratore servente, inoltre gli attributi sono separati attraverso una barra verticale ('**|**').

```
COPY Indirizzi FROM STDIN;
```

In questo caso, si aggiungono righe alla tabella '**Indirizzi**', utilizzando quanto proviene dallo standard input (alla fine deve apparire la sequenza di escape '**\.**').

```
COPY Indirizzi FROM '/tmp/prova' USING DELIMITERS '|';
```

Si aggiungono righe alla tabella '**Indirizzi**', utilizzando quanto proviene dal file '/tmp/prova', che si trova nel file system dell'elaboratore servente. In particolare, gli attributi sono separati da una barra verticale ('|').

325.7.2 Riorganizzazione del contenuto di una base di dati

Nel capitolo precedente si è già accennato all'istruzione '**VACUUM**', con la quale si riorganizzano i dati, eliminando i resti di una transazione interrotta, ricostruendo gli indici e le statistiche interne. Se non si ha la pretesa di analizzare la base di dati, lo schema sintattico è molto semplice:

```
VACUUM [ tabella ]
```

Se non si indica una tabella particolare, si intende intervenire su tutta la base di dati su cui si sta lavorando.

È conveniente utilizzare questa istruzione tutte le volte che si annulla una transazione.

Per poter eseguire le operazioni relative all'istruzione '**VACUUM**', è necessario un blocco esclusivo delle tabelle coinvolte. Questo blocco è rappresentato in pratica da un file collocato nella directory che contiene i file della base di dati relativa. Il file si chiama 'pg_vlock'; se si interrompe in qualche modo un'istruzione '**VACUUM**', questo file non viene rimosso, impedendo tutte le attività sulla base di dati. Se questa situazione dovesse verificarsi, si può disattivare il programma servente, in modo da essere certi che non ci sia alcun accesso ai dati, così da poter eliminare successivamente il file che rappresenta questo blocco.

325.7.3 Impostazione dell'ora locale

L'SQL dispone dell'istruzione '**SET TIME ZONE**' per definire l'ora locale e di conseguenza lo scostamento dal tempo universale. PostgreSQL dispone della stessa istruzione che funziona in modo molto simile allo standard; per la precisione, la definizione dell'ora locale avviene attraverso le definizioni riconosciute dal sistema operativo (nel caso di GNU/Linux si tratta delle definizioni che si articolano a partire dalla directory '/usr/share/zoneinfo/').

```
SET TIME ZONE { 'definizione_ora_locale' | LOCAL }
```

Per esempio, per definire che si vuole fare riferimento all'ora locale italiana, si potrebbe usare il comando seguente:

```
SET TIME ZONE 'Europe/Rome';
```

Questa impostazione riguarda la visione del programma cliente, mentre il programma servente può essere stato preconfigurato attraverso le variabili di ambiente '**LC_***' oppure la variabile '**LANG**', che in questo caso hanno effetto sullo stile di rappresentazione delle informazioni data-orario. Anche il programma cliente può essere preconfigurato attraverso la variabile di ambiente '**PGTZ**', assegnandole gli stessi valori che si possono utilizzare per l'istruzione '**SET TIME ZONE**'.

325.8 Riferimenti

- *sql(l)*
- The PostgreSQL Development Team, *PostgreSQL User's Guide*

  <file:///usr/share/doc/postgresql/user/index.html>

  <file:///usr/share/doc/postgresql/user.ps>*

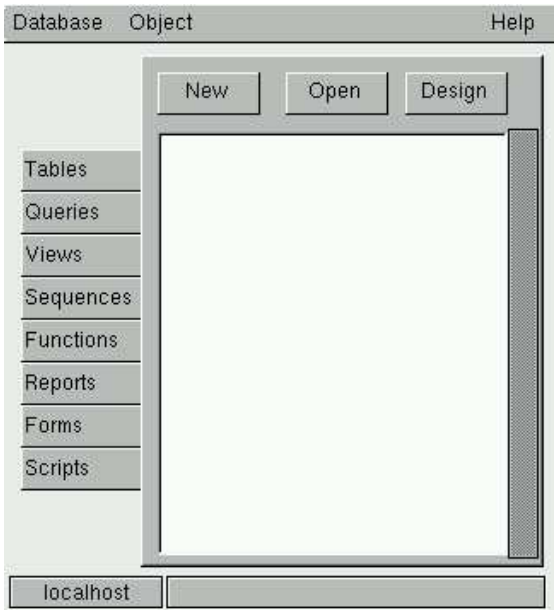
PostgreSQL: accesso attraverso PgAccess

PgAccess [1] è un componente di una libreria Tcl/Tk: LibPgTcl. A volte viene distribuito come un pacchetto autonomo, che comunque dipende dalla libreria indicata, oppure viene incluso nello stesso pacchetto della libreria. PgAccess è un programma frontale (che utilizza l'interfaccia grafica) per accedere alle funzionalità di PostgreSQL.

> Prima di poter utilizzare qualunque programma frontale per PostgreSQL, occorre ricordare di configurare correttamente PostgreSQL stesso, in modo che questo consenta gli accessi previsti.

PgAccess è costituito in pratica dall'eseguibile '**`pgaccess`**', che si utilizza senza argomenti e si presenta inizialmente come si vede nella figura 326.1.

Figura 326.1. Finestra iniziale di PgAccess, quando viene avviato per la prima volta dall'utente.



Mentre lo si usa, PgAccess memorizza alcune informazioni nel file '`~/.pgaccessrc`' e questo fatto facilita successivamente le operazioni di accesso alla base di dati da parte dell'utente.

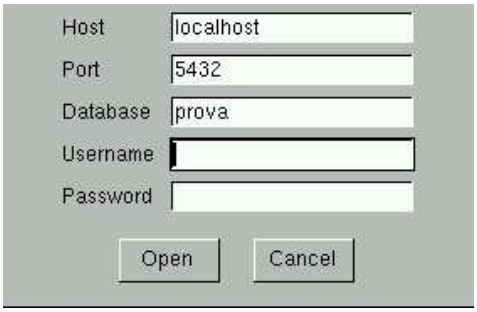
Contrariamente a quello che ci si potrebbe aspettare, PgAccess è un programma frontale realizzato con molta cura e molto potente, che permette di sfruttare bene le potenzialità di PostgreSQL. Purtroppo è un po' difficile spiegare nel dettaglio il suo funzionamento; pertanto, lo scopo di questo capitolo è solo quello di permettere all'utilizzatore di cominciare e di sapere come continuare. Con l'uso, i particolari del suo funzionamento dovrebbero rivelarsi senza troppi problemi.

326.1 Accesso alla base di dati

PostgreSQL è un DBMS in grado di gestire diverse basi di dati simultaneamente; pertanto, con PgAccess è necessario stabilire per prima cosa quale sia la base di dati. Dal menù '**`Database`**', si seleziona la funzione '**`Open`**', ottenendo la mascherina che si vede nella figura 326.2. Da lì si possono indicare tutte le informazioni necessarie alla connessione con la base di dati desiderata; in particolare, per quanto riguarda le informazioni sull'autenticazione, queste sono richieste solo in base al modo in cui sono stati regolati i permessi di accesso da parte di PostgreSQL.

[1] **PgAccess** software libero con licenza speciale

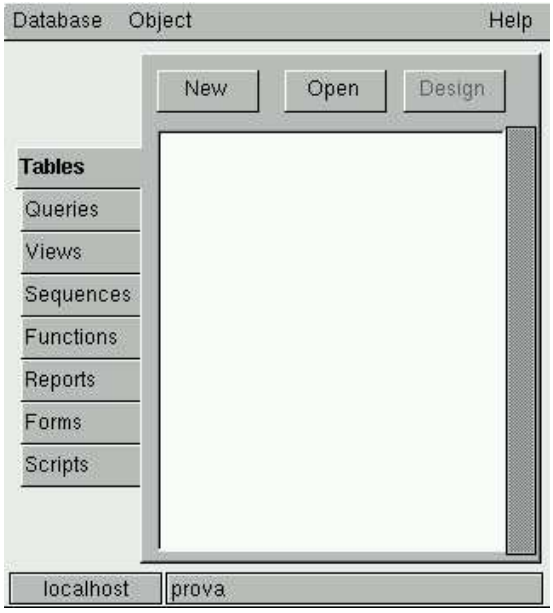
Figura 326.2. Connessione alla base di dati '**prova**', presso il nodo locale, utilizzando l'autenticazione predefinita.



Attraverso PgAccess non è possibile creare una base di dati. Per questo occorre usare il comando '**createdb**' di PostgreSQL, descritto nel capitolo 324.

La base di dati aperta, assieme all'indicazione del nodo presso la quale si trova il DBMS con cui si interagisce, appare in basso, nella finestra principale di PgAccess.

Figura 326.3. Quando è attiva una connessione con una base di dati, lo si vede dalle informazioni che appaiono in basso nella finestra principale di PgAccess.



È importante ricordare che PgAccess tiene nota dell'ultima base di dati aperta attraverso il file di configurazione '`~/.pgaccessrc`'; in questo modo la connessione viene ritentata automaticamente all'avvio del programma la volta successiva che lo si utilizza. Tuttavia, questo particolare del funzionamento di PgAccess può essere configurato attraverso la funzione *`Preferences`* del menù *`Database`*.

326.2 Gli «oggetti» secondo PgAccess

Dal punto di vista di PgAccess, una base di dati contiene degli «oggetti» (secondo la stessa filosofia di PostgreSQL). Questi possono essere delle tabelle, il risultato di interrogazioni SQL, delle viste, delle stampe, ecc.

Per intervenire su ognuno di questi oggetti basta selezionare la voce relativa che si trova sulla parte sinistra (nella figura 326.3 si vede selezionata la gestione delle tabelle). Nel riquadro centrale

c'è lo spazio per elencare i nomi degli oggetti di quel tipo che risultano presenti, mentre sopra appaiono alcuni pulsanti grafici che si riferiscono alle cose che si possono fare con tali oggetti.

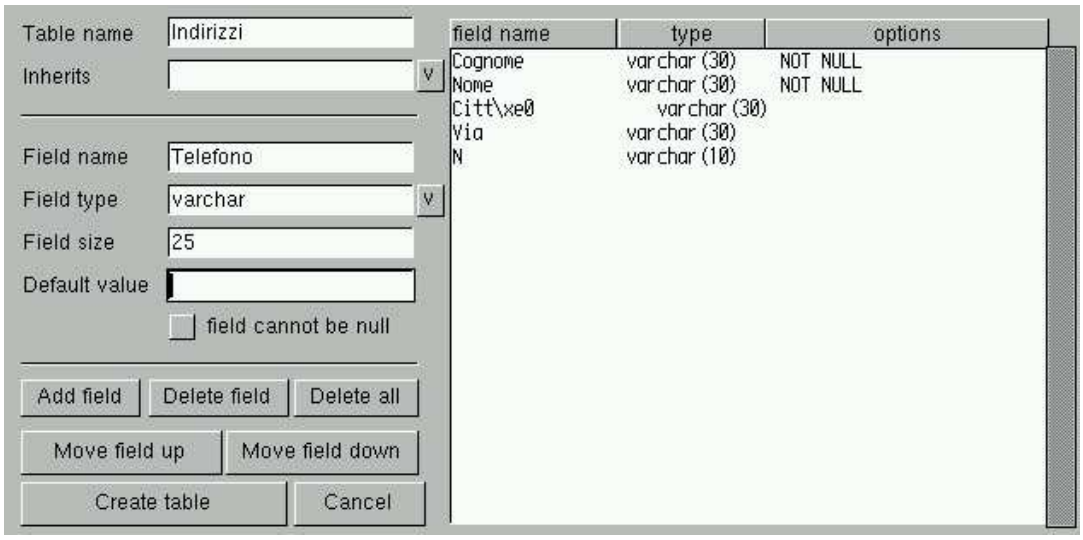
Evidentemente, il pulsante grafico [NEW] serve a creare un oggetto nuovo, [OPEN] serve ad accedervi e [DESIGN] serve a modificarne la struttura (ammesso che ciò sia consentito in base al tipo di oggetto). Tuttavia, il menù *Object* offre altre possibilità, per esempio la modifica del nome dell'oggetto e la visualizzazione della sua struttura.

> PgAccess gestisce una serie di oggetti aggiuntiva rispetto a quanto fa PostgreSQL. Per realizzarli, PgAccess gestisce delle tabelle aggiuntive che non vengono mostrate all'utente, distinguibili per il fatto di avere un nome che inizia per '**pga**_'. In generale, queste tabelle hanno tutti i permessi di accesso per tutti gli utenti di PostgreSQL.

326.3 Tabelle

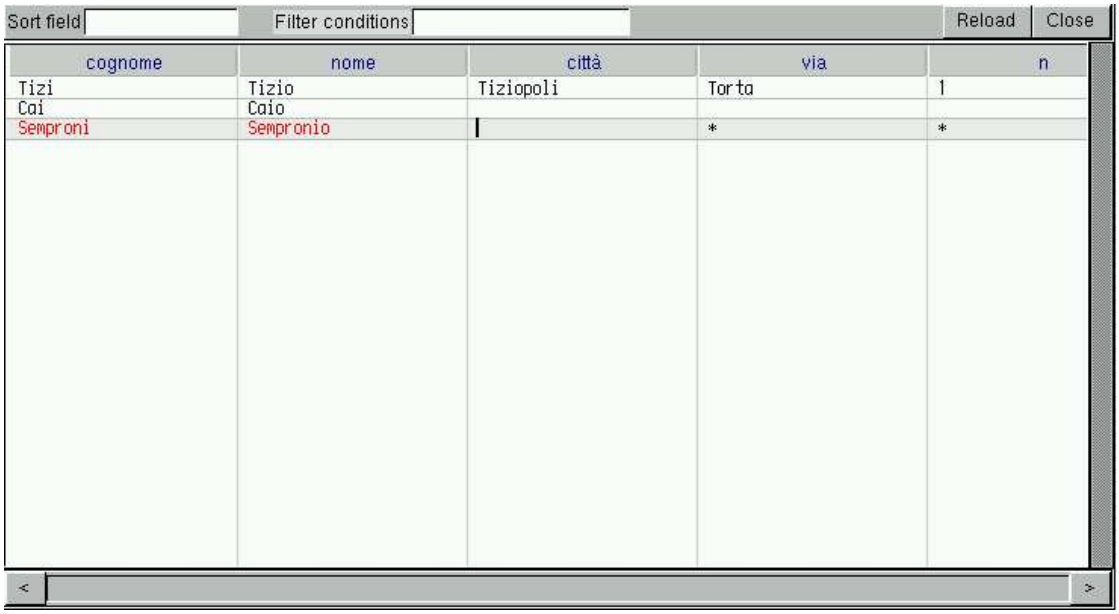
La figura 326.4 mostra l'esempio della creazione di una tabella molto semplice, per contenere una serie di indirizzi. In particolare si può osservare il fatto che PgAccess abbia convertito opportunamente la lettera «à» nella sequenza '**\xe0**'. Alla creazione della tabella, dopo avere selezionato la voce relativa a questo tipo di oggetto, si accede selezionando il pulsante grafico [NEW].

Figura 326.4. Finestra per la creazione di una tabella.



Una volta creata la tabella (si ottiene questo confermando con il pulsante grafico [CREATE TABLE]), il suo nome appare nella parte centrale della finestra principale del programma; per accedere al suo contenuto basta selezionare il pulsante grafico [OPEN], ottenendo così una tabella di scorrimento con la quale si possono aggiungere e modificare righe preesistenti. La figura 326.5 mostra l'inserimento di alcuni nomi. Si osservi in particolare il fatto che, eventualmente, si può richiedere espressamente l'aggiunta di una riga nuova premendo il terzo tasto del mouse.

Figura 326.5. Finestra per lo scorrimento del contenuto di una tabella.

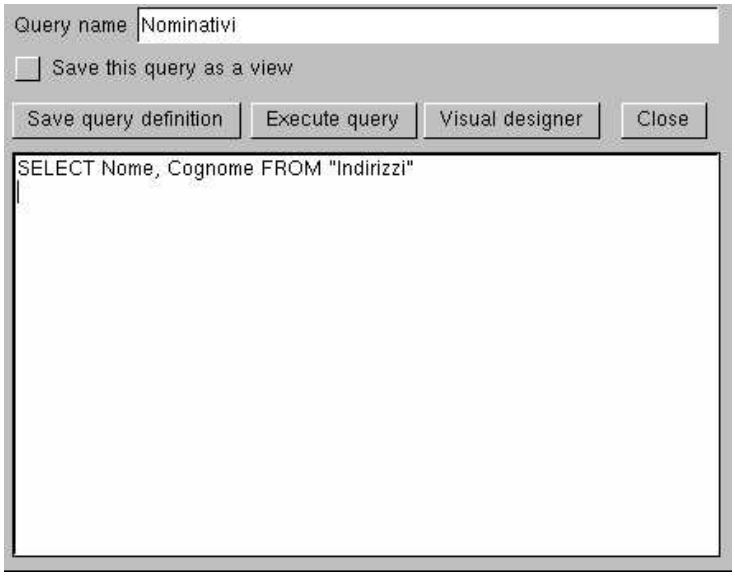


Vale la pena di osservare che la maschera di scorrimento e inserimento dati nella tabella, permette di leggere le righe in ordine, in base a una certa colonna, filtrando eventualmente le righe in base a una condizione. Si stabilisce questo mettendo il nome di una colonna nella casella '**`Sort field`**' e mettendo l'espressione della condizione di filtro nella casella '**`Filter conditions`**': se poi si seleziona il pulsante grafico RELOAD, si riottiene il contenuto ordinato e filtrato in base alle preferenze indicate.

326.4 Interrogazioni e viste

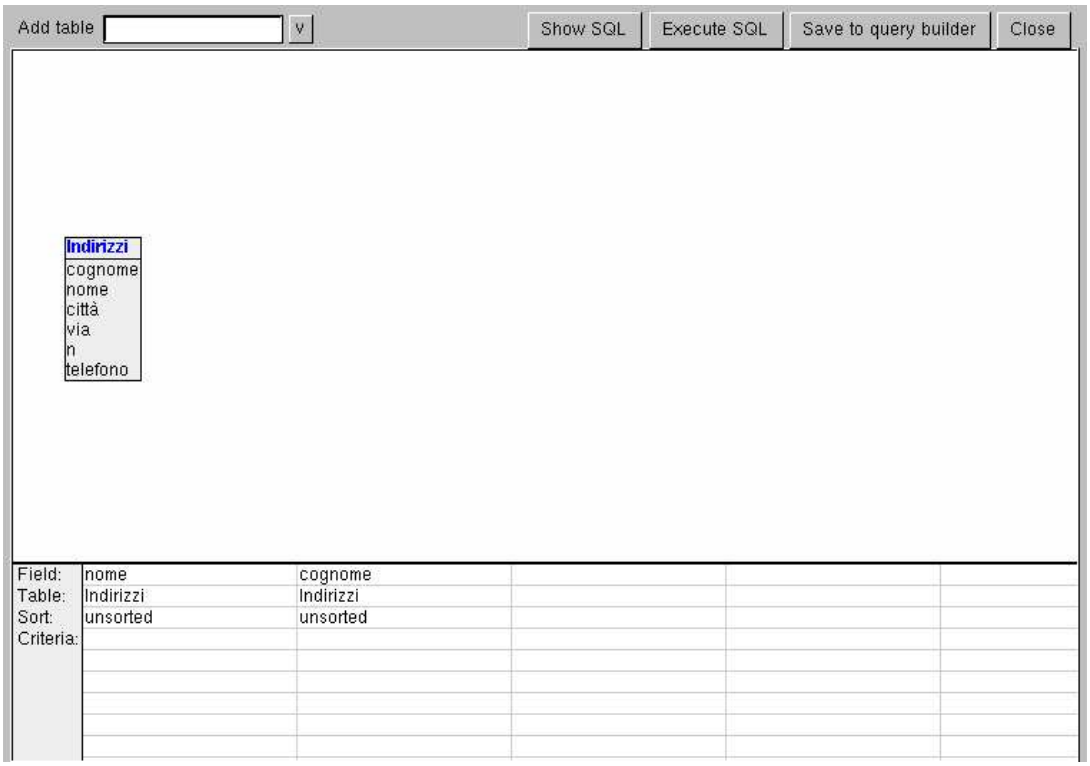
È possibile realizzare facilmente dei modelli di interrogazione e delle viste, attraverso la selezione delle voci QUERIES e VIEWS. Nel primo caso si tratta di interrogazioni SQL che vengono memorizzate da PgAccess e richiamate a piacere, mentre nel secondo si tratta di viste vere e proprie. A livello operativo, con PgAccess le due cose sono praticamente identiche, per cui si passa generalmente per la creazione di un'interrogazione SQL che poi, eventualmente, si salva come vista. La figura 326.6 mostra la definizione dell'interrogazione '**`Nominativi`**', abbinata al comando '**`SELECT Cognome, Nome FROM "Indirizzi"`**', scritto manualmente dall'utilizzatore.

Figura 326.6. Finestra per la creazione di un'interrogazione.



Nella figura si può osservare che è disponibile una casella di selezione attraverso la quale si può richiedere di salvare come vista. In particolare, con il pulsante grafico SAVE QUERY DEFINITION si salva il modello dell'interrogazione, con il nome fissato in alto; ma volendo, con il pulsante grafico VISUAL DESIGNER, si accede a una maschera per la definizione grafica dell'interrogazione, come si vede nella figura 326.7.

Figura 326.7. Finestra per la creazione visuale di un'interrogazione.



In alto appare una casella in cui si deve indicare il nome di una tabella da cui si vogliono prelevare i campi; una volta fatto, appare un riepilogo di questi campi, in un riquadro. Questi nomi possono essere trascinati con il puntatore del mouse, in basso, dove vengono elencati i campi da includere nell'interrogazione; se si sbaglia, gli elementi che si vogliono togliere possono essere cancellati premendo il tasto [*Canc*] ([*Del*] nelle tastiere inglesi). Nella figura mostrata, sono già

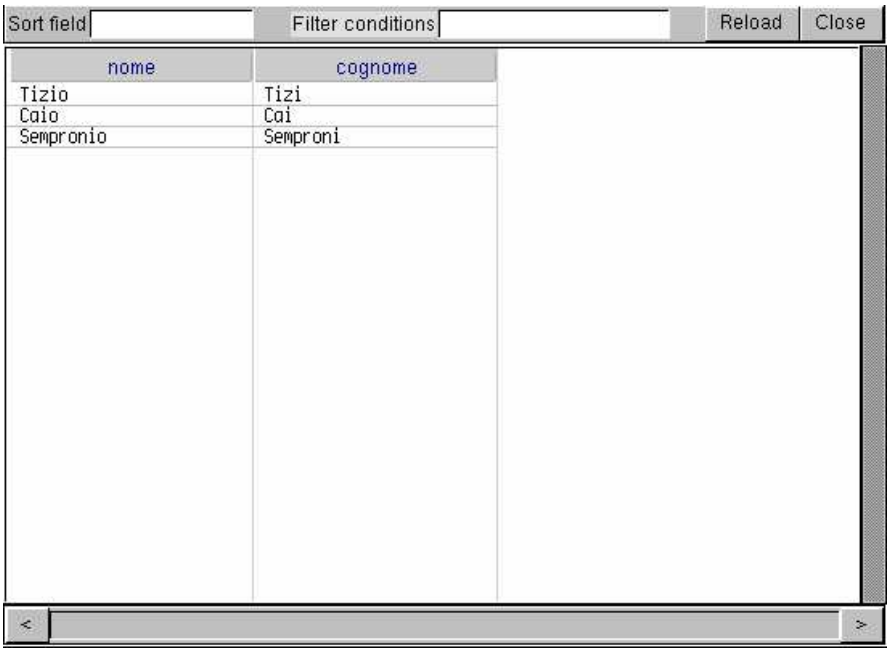
stati trascinati e depositati i campi del nome e del cognome.

Al termine, se si è soddisfatti del risultato, si può confermare con il pulsante grafico SAVE TO QUERY BUILDER, ritrovando poi nella finestra precedente l'interrogazione corrispondente alle scelte fatte, che può essere ritoccata a mano se lo si desidera. Nel caso dell'esempio mostrato, l'interrogazione SQL che si ottiene è:

```
select t0.nome, t0.cognome from "Indirizzi" t0
```

L'apertura di un'interrogazione o di una vista, genera lo scorrimento del risultato dell'interrogazione, oppure della vista, come si vede nella figura 326.8 che fa sempre riferimento agli esempi precedenti.

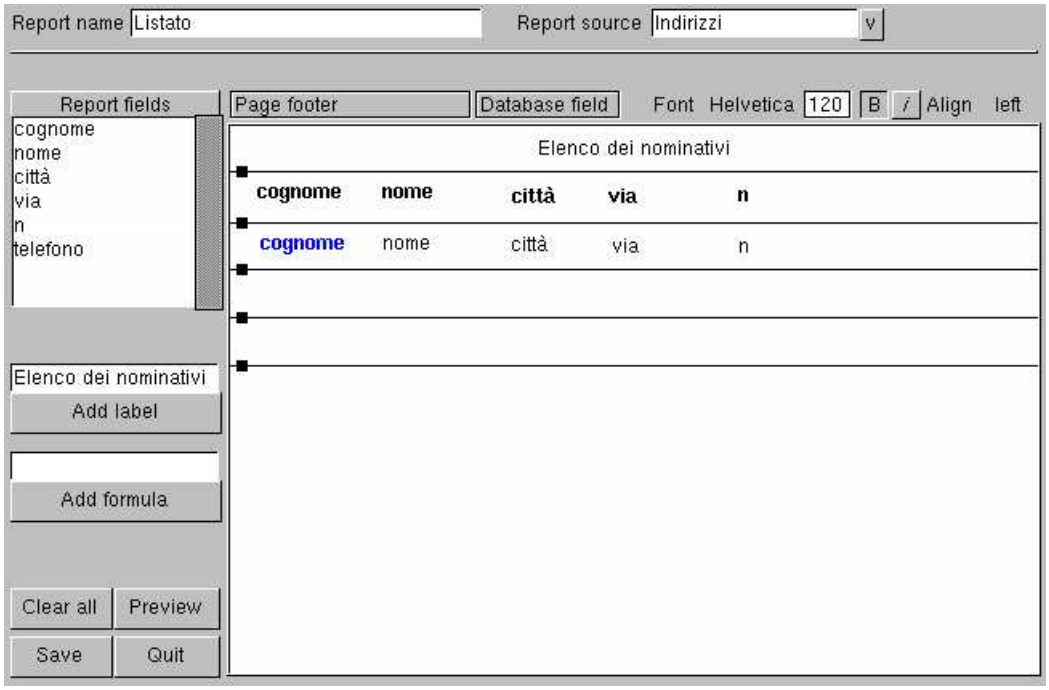
Figura 326.8. Scorrimento di una vista.



326.5 Stampe

Con PgAccess è possibile definire anche delle stampe, nel senso di rapporti stampati contenenti il risultato di un'interrogazione SQL. La figura 326.9 mostra la finestra che si utilizza per questo scopo, dove è già iniziata la compilazione dello schema di stampa.

Figura 326.9. Creazione di un rapporto.

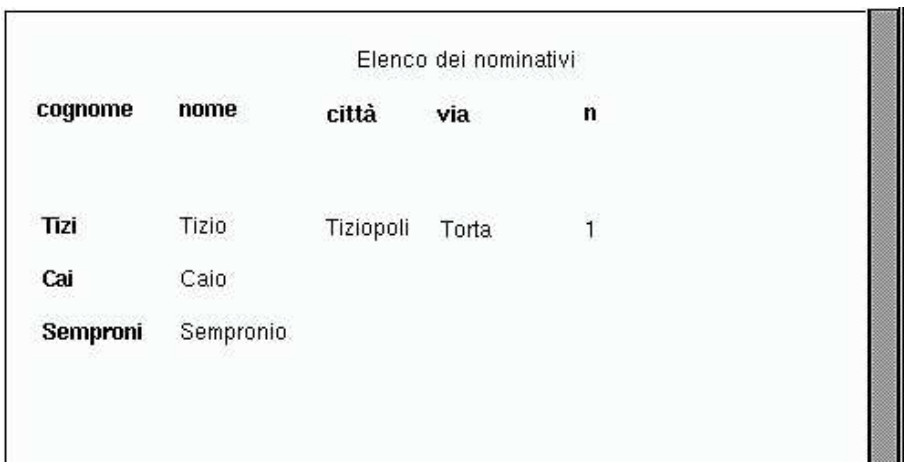


Una volta selezionata la tabella da cui prelevare i campi, dopo aver indicato il nome del rapporto che si vuole generare, basta fare un clic con il tasto sinistro del mouse mentre si punta sul nome del campo che si vuole inserire sullo schema di destra (che rappresenta il modello della stampa). Una volta che sono apparsi i nomi nello spazio a destra, questi possono essere trascinati dove si vuole, eventualmente possono anche essere cancellati usando il tasto [*Canc*]. Nell'esempio della figura, si vede anche che è stato inserito un titolo.

Spostando il puntatore del mouse sullo spazio che rappresenta lo schema di stampa, si vede cambiare la sua descrizione in alto. Nella figura mostrata viene indicato '**`Page footer`**', perché in quel momento il puntatore del mouse era nella penultima riga di quello schema.

Per verificare il risultato, è disponibile anche un'anteprima, che si ottiene selezionando il pulsante grafico PREVIEW. Seguendo gli esempi precedenti, la figura 326.10 mostra questa anteprima. Da lì si può passare alla stampa, che però potrebbe limitarsi a generare un file PostScript.

Figura 326.10. Anteprima di stampa.



326.6 Riferimenti

- *PgAccess*

  <http://www.flex.ro/pgaccess/>

PostgreSQL: accesso attraverso WWW-SQL

WWW-SQL [1] è un programma CGI in grado di creare pagine HTML a partire dalle informazioni ottenute da una base di dati PostgreSQL o MySQL. In questo capitolo si vuole vedere in particolare l'interazione rispetto alle basi di dati di PostgreSQL. In ogni caso, per poter leggere questo capitolo, occorre sapere cosa sia un programma CGI e come interagisce con un servente HTTP, come spiegato a partire dal capitolo 162. La scelta di collocare qui queste informazioni, è dovuta al fatto che si tratta di un argomento legato alla programmazione SQL; inoltre, WWW-SQL è praticamente un interprete di un linguaggio specifico, che gli permette di definire le richieste alla base di dati e di generare il risultato finale desiderato.

È molto probabile che la propria distribuzione GNU/Linux abbia organizzato due pacchetti distinti, in base all'uso che se ne intende fare, per l'abbinamento con PostgreSQL, oppure con MySQL. In questo modo, il nome del programma CGI a cui si deve fare riferimento può cambiare leggermente, anche da una distribuzione all'altra. Qui si fa riferimento al nome '**`www-pgsql`**' per quello che riguarda l'uso con PostgreSQL.

327.1 Principio di funzionamento

Ammesso che il pacchetto organizzato dalla propria distribuzione sia stato realizzato nel modo corretto, l'eseguibile '**`www-pgsql`**' dovrebbe trovarsi nella directory più adatta per i programmi CGI, ovvero quella a cui si accede normalmente con l'URI *`http://localhost/cgi-bin/`*. In tal caso, per accedere a questo programma, basta avviare il proprio navigatore preferito e puntare sull'indirizzo *`http://localhost/cgi-bin/www-pgsql`*. Ma non basta, dal momento che il programma in questione ha bisogno di interpretare un file HTML speciale dal quale restituisce poi un risultato. Per capire come funziona la cosa, prima ancora di avere affrontato lo studio del linguaggio specifico di WWW-SQL, si può provare con un file HTML normale: si supponga di avere a disposizione il file '`http://localhost/index.html`'; per fare in modo che WWW-SQL lo analizzi, basta indicare l'URI *`http://localhost/cgi-bin/www-pgsql/index.html`*. Il risultato è identico all'originale, ma per arrivare a questo si passa attraverso l'elaborazione del programma CGI, dimostrando così il suo funzionamento.

Volendo, se il proprio programma servente HTTP è Apache, è possibile rendere la cosa più elegante attraverso una configurazione opportuna del file '`srm.conf`' (si veda a questo proposito il capitolo 160 su Apache). Per esempio si potrebbe fare in modo che i file che terminano con l'estensione '`.pgsql`' vengano elaborati automaticamente attraverso il programma CGI in questione:

```
Action www-pgsql /cgi-bin/www-pgsql
AddHandler www-pgsql pgsql
```

Tuttavia, occorre considerare che alcune installazioni di Apache sono state predisposte in modo da impedire l'utilizzazione dell'istruzione '**`Action`**'. Se dopo le modifiche di questo file, il servizio di Apache non si riavvia, ciò potrebbe essere un sintomo di questo problema.

[1] **WWW-SQL** GNU GPL

327.2 Preparazione delle basi di dati e accesso

Perché il programma CGI possa accedere alle basi di dati di PostgreSQL, occorre ricordare di predisporre gli utenti e i permessi necessari all'interno della gestione delle basi di dati stesse. Di solito, si prevede la possibilità di accesso per l'utente '**nobody**', dal momento che il servente viene avviato solitamente con i privilegi di questo utente e, di conseguenza, il programma CGI eredita la stessa personalità.[2] Vale la pena di ricordare che per PostgreSQL occorre aggiungere l'utente '**nobody**' nel modo seguente:

```
postgres$ createuser nobody
```

Quindi occorre intervenire nelle basi di dati regolando i permessi attraverso i comandi '**GRANT**' e '**REVOKE**', tenendo conto che a questo proposito si può consultare quanto già spiegato nel capitolo 324. Per fare un esempio, volendo concedere l'accesso in lettura alla tabella '**Indirizzi**', della base di dati '**anagrafe**', all'utente '**nobody**', si potrebbe agire come si vede di seguito:

```
postgres$ psql anagrafe[ Invio ]

anagrafe=> GRANT SELECT ON Indirizzi TO nobody;[ Invio ]
```

Tuttavia, occorre tenere in considerazione il fatto che non tutte le distribuzioni GNU/Linux sono organizzate in modo che i privilegi di funzionamento del programma servente del servizio HTTP siano pari a quelli dell'utente '**nobody**'. Per esempio, nel caso della distribuzione GNU/Linux Debian, viene usato l'utente apposito '**www-data**': questo nome particolare, non può essere inserito negli utenti di PostgreSQL, a causa del fatto che contiene un trattino ('**-**'), per cui, occorre agire in modo differente, ma questo verrà descritto in seguito, in occasione della descrizione dell'istruzione '**<! SQL CONNECT >**'.

327.3 Linguaggio di WWW-SQL

WWW-SQL interpreta un file HTML alla ricerca di istruzioni secondo il formato schematizzato di seguito:

```
<! SQL comando [argomento ...] >
```

Come si vede, queste istruzioni assomigliano a dei commenti per l'HTML, ma anche se non lo sono realmente, di solito i navigatori ignorano dei marcatori di questo tipo. Tuttavia, questa si può considerare solo come una misura di sicurezza, dal momento che questi file non dovrebbero essere raggiunti direttamente, ma solo attraverso l'intermediazione di WWW-SQL.

Le istruzioni di WWW-SQL rappresentano un linguaggio di programmazione, semplice, ma efficace per lo scopo che ci si prefigge. Si osservi che il «comando» è una parola chiave che rappresenta il tipo di azione che si intende svolgere; inoltre, gli argomenti possono essere presenti o meno, in funzione del comando. Gli argomenti di un comando possono essere racchiusi tra apici doppi ('**"**...**"**'): all'interno di queste stringhe si possono indicare delle variabili da espandere e si possono usare anche delle sequenze di escape per rappresentare simboli speciali che altrimenti avrebbero un altro significato.

> Le parole chiave che costituiscono le istruzioni di WWW-SQL possono essere scritte indipendentemente utilizzando lettere maiuscole o minuscole. Inoltre, lo spazio dopo il delimitatore iniziale '**<!**' e lo spazio prima del delimitatore finale '**>**' sono facoltativi.

[2] Naturalmente, lo stesso discorso può valere per un utente fittizio diverso, realizzato appositamente per il controllo della gestione del servizio HTTP.

Per iniziare subito con un esempio che faccia capire la logica di funzionamento di WWW-SQL, si osservi il «programma» seguente, rappresentato dal file '`variabili.pgsql`':

```
<HTML>
<HEAD>
    <TITLE>Esempio sul funzionamento delle variabili con WWW-SQL</TITLE>
</HEAD>
<BODY>
<H1>Esempio sul funzionamento delle variabili con WWW-SQL</H1>

<P><! SQL PRINT "var = $var" ></P>

<p><FORM ACTION="variabili.pgsql" METHOD="GET">
    <INPUT NAME="var">
    <INPUT TYPE="submit">
</form></p>

</BODY>
```

L'unica istruzione per WWW-SQL è '**`<!SQL PRINT`**`…>`', con la quale si vuole ottenere la visualizzazione di una stringa tra apici doppi. Si osservi che '**`$var`**' è il riferimento alla variabile '**`var`**', che viene espanso, come parte della valutazione della stringa.

Come si può intuire leggendo l'esempio, i campi definiti attraverso i modelli (gli elementi '**`FORM`**'), si traducono in variabili per WWW-SQL.

Per verificare il funzionamento di questo programma, supponendo di avere collocato il file '`variabili.pgsql`' nella directory iniziale dei documenti HTML offerti dal servente HTTP, basta puntare il navigatore sull'indirizzo *`http://localhost/cgi-bin/www-pgsql/variabili.pgsql`* (sempre ammettendo che l'indirizzo *`http://localhost/cgi-bin/www-pgsql`* corrisponda all'avvio del programma CGI che costituisce in pratica WWW-SQL).

Quello che si ottiene dovrebbe essere un modulo HTML molto semplice, dove si può inserire un testo. Inviando il modulo compilato, dovrebbe essere restituito lo stesso modulo, con la stringa iniziale aggiornata, dove viene mostrato che è stato recepito il dato inserito (nella figura 327.1 si vede che era stata inviata la stringa «Saluti».

Figura 327.1. Risultato dell'interpretazione del file '`variabili.pgsql`' attraverso WWW-SQL.

I sorgenti di WWW-SQL possono essere compilati in modo differente. In particolare, si può distinguere tra due tipi di scansione: il tipo vecchio non permette l'uso di istruzioni che prevedono un'iterazione. In pratica, in quel caso, non funzionano i cicli iterativi e gli altri comandi correlati.

327.3.1 Espressioni

Si distinguono due tipi di espressioni che si possono valutare all'interno delle istruzioni di WWW-SQL: quelle che si applicano ai valori numerici e quelle che si applicano alle stringhe. Le tabelle 327.1 e 327.2 elencano gli operatori che possono essere utilizzati a questo proposito. Si osservi in particolare l'operatore '**:**', che permette di fare un confronto tra una stringa e un'espressione regolare.

Tabella 327.1. Elenco degli operatori utilizzabili con operandi numerici.

Operatore e operandi	Descrizione
+*op*	Non ha alcun effetto.
-*op*	Inverte il segno dell'operando.
op1 + *op2*	Somma i due operandi.
op1 - *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Modulo -- il resto della divisione tra il primo e il secondo operando.
op1 ^ *op2*	Eleva il primo operando alla potenza del secondo.
op1 == *op2*	*Vero* se gli operandi sono uguali.
op1 = *op2*	*Vero* se gli operandi sono uguali (sinonimo di '**==**').
op1 != *op2*	*Vero* se gli operandi sono differenti.
op1 > *op2*	*Vero* se il primo operando è maggiore del secondo.
op1 < *op2*	*Vero* se il primo operando è minore del secondo.
op1 >= *op2*	*Vero* se il primo operando è maggiore o uguale al secondo.
op1 <= *op2*	*Vero* se il primo operando è minore o uguale al secondo.
! *op*	Negazione logica.
op1 && *op2*	AND logico.
op1 & *op2*	AND logico (sinonimo di '**&&**').
op1 || *op2*	OR logico.
op1 | *op2*	OR logico (sinonimo di '**||**').

Tabella 327.2. Elenco degli operatori utilizzabili con operandi di tipo stringa.

Operatore e operandi	Descrizione
op1 == *op2*	*Vero* se gli operandi sono uguali.
op1 != *op2*	*Vero* se gli operandi sono differenti.
op1 > *op2*	*Vero* se il primo operando è lessicograficamente successivo al secondo.
op1 < *op2*	*Vero* se il primo operando è lessicograficamente precedente al secondo.
op1 >= *op2*	*Vero* se il primo operando non è lessicograficamente precedente al secondo.
op1 <= *op2*	*Vero* se il primo operando non è lessicograficamente successivo al secondo.
str : *regexp*	*Vero* se l'espressione regolare corrisponde alla stringa.

All'interno delle stringhe è prevista l'espansione di variabili e sono anche riconosciute alcune sequenze di escape (tabella 327.3). Le variabili in questione vanno intese come parte del linguaggio di WWW-SQL; alcune di queste sono la ripetizione di variabili di ambiente corrispondenti, altre sono variabili interne del programma (come elencato nella tabella 327.4), altre ancora possono essere definite all'interno del «programma» stesso, o meglio ancora, attraverso dei moduli, come è stato mostrato nell'esempio iniziale. Le variabili vengono riconosciute in quanto scritte secondo lo schema seguente:

prefissonome_della_variabile

Il prefisso è un simbolo a scelta tra: '**$**', '**@**', '**?**', '**#**'. In pratica, '**$var**', '**@var**', '**?var**', e '**#var**', sono riferimenti identici alla stessa variabile '**var**'. Per questo motivo, se si vogliono usare i simboli corrispondenti a questi prefissi in modo letterale, occorre usare una sequenza di escape.

Tabella 327.3. Sequenze di escape utilizzabili all'interno delle stringhe.

Escape	Significato
\\	\
\"	"
\n	<*LF*>
\t	<*HT*> (tabulazione)
\$	$
\@	@
\#	#
\?	?
\~	~

Tabella 327.4. Variabili interne di WWW-SQL.

Variabile	Descrizione
AFFECTED_ROWS	Numero di righe coinvolte dall'ultima interrogazione.
NUM_FIELDS	Numero di campi restituiti dall'ultima interrogazione.
NUM_ROWS	Numero di righe restituiti dall'ultima interrogazione.
WWW_SQL_VERSION	Versione di WWW-SQL.
GATEWAY_INTERFACE	Versione dell'interfaccia CGI.
HOSTTYPE	Tipo di macchina del servente HTTP.
HTTPHOST	Nome del nodo servente.
HTTP_REFERER	Pagina da cui proviene il cliente.
HTTP_USER_AGENT	Nome del programma di navigazione (cliente).
OSTYPE	Nome del sistema operativo del servente.
PATH_INFO	Percorso relativo dello script attuale.
PATH_TRANSLATED	Percorso assoluto del file corrispondente allo script attuale.
REMOTE_ADDR	Indirizzo del nodo remoto.
REMOTE_HOST	Nome del nodo remoto.
SERVER_ADMIN	Indirizzo di posta elettronica dell'amministratore.
SERVER_NAME	Nome del servente.
SERVER_PORT	Numero della porta utilizzata per la connessione con il servente.
SERVER_PROTOCOL	Nome e versione del protocollo (HTTP).
SERVER_SOFTWARE	Nome del software usato come servente HTTP.
SCRIPT_FILENAME	Percorso del programma CGI (l'eseguibile di WWW-SQL).
SCRIPT_NAME	Percorso relativo del programma CGI (l'eseguibile di WWW-SQL).
REQUEST_URI	Indirizzo richiesto.

Per prendere confidenza con le variabili interne di WWW-SQL, si può realizzare lo script seguente ('`interne.pgsql`'), che con l'istruzione '**`<!SQL DUMPVARS>`**' le elenca tutte. La figura 327.2 mostra il risultato che si potrebbe ottenere.

```
<HTML>
<HEAD>
    <title>Visualizzazione delle variabili interne</title>
</HEAD>
<BODY>
<H1>Visualizzazione delle variabili interne</H1>

<! SQL DUMPVARS >

</BODY>
```

Figura 327.2. Esempio del contenuto delle variabili interne attraverso l'istruzione '**<!SQL DUMPVARS>**'.

```
                          Visualizzazione delle variabili interne

        WWW_SQL_VERSION = 0.5.5
        SERVER_SOFTWARE = Apache/1.3.3 (Unix) Debian/GNU
        SERVER_PROTOCOL = HTTP/1.0
        SERVER_PORT = 80
        SERVER_NAME = dinkel.brot.dg
        SERVER_ADMIN = webmaster@dinkel.brot.dg
        SCRIPT_FILENAME = /usr/lib/cgi-bin/www-pgsql
        SCRIPT_NAME = /cgi-bin/www-pgsql
        REQUEST_URI = /cgi-bin/www-pgsql/interne.pgsql
        REMOTE_ADDR = 127.0.0.1
        QUERY_STRING =
        PATH_TRANSLATED = /var/www/interne.pgsql
        PATH_INFO = /interne.pgsql
        HTTP_USER_AGENT = Lynx/2.8.1rel.2 libwww-FM/2.14
        HTTP_HOST = localhost
        GATEWAY_INTERFACE = CGI/1.1
        DOCUMENT_ROOT = /var/www
```

327.3.2 Strutture di controllo

Attraverso le istruzioni di WWW-SQL, si possono realizzare le strutture di controllo che sono comuni nei linguaggi di programmazione. È prevista la struttura condizionale e il ciclo iterativo.

```
<! SQL IF espressione >
    …
[<! SQL ELSIF espressione >]
    …
[<! SQL ELSE >]
    …
<! SQL ENDIF >
```

La struttura condizionale che si vede nello schema, permette di delimitare uno spazio da filtrare in base all'esito delle espressioni condizionali coinvolte. Per esempio,

```
<! SQL IF $NUM_ROWS == 10 >
    <P>Il numero delle righe è uguale a 10.</P>
<! SQL ELSE >
    <P>Il numero delle righe non corrisponde a
    quanto previsto.</P>
<! SQL ENDIF >
```

in questo modo si condiziona la visualizzazione di una frase in base al fatto che la variabile '**NUM_ROWS**' contenga o meno il valore 10.

È importante osservare che l'espressione usata come condizione di controllo potrebbe restituire un risultato numerico e non logico. In tal caso, lo zero corrisponde a *Falso*, mentre qualunque altro valore corrisponde a *Vero*.

```
<! SQL WHILE espressione >
    …
<! SQL DONE >
```

La struttura iterativa che si vede nello schema, permette di delimitare uno spazio da interpretare ripetitivamente, finché l'espressione condizionale introduttiva continua a restituire il valore *Vero* (o un valore numerico diverso da zero).

```
<! SQL SET contatore 10 >
<! SQL WHILE $contatore > 0 >
```

```
    <P>Il contatore ha raggiunto il livello <! SQL PRINT "$contatore" >.</P>
    <! SQL SETEXPR contatore $contatore - 1 >
<! SQL DONE >
```

L'esempio mostra l'inizializzazione di una variabile, denominata '**`contatore`**', al valore iniziale 10; quindi inizia un ciclo iterativo che si arresta quando tale variabile raggiunge lo zero. A ogni ciclo, viene visualizzato il contenuto della variabile, che subito dopo viene ridotto di un'unità.[3]

Nell'ambito di un'iterazione, possono essere usate delle istruzioni per interrompere il ciclo in corso o per interrompere tutta l'iterazione:

```
<! SQL CONTINUE >
<! SQL BREAK >
```

La prima della due istruzioni interrompe il ciclo attuale, facendo riprendere immediatamente l'iterazione, mentre il secondo interrompe l'iterazione del tutto.

Esiste anche un altro tipo di iterazione, il cui scopo è la scansione delle righe ottenute dall'interrogazione di una base di dati:

```
<! SQL PRINT_LOOP riferimento_all'interrogazione >
    …
<! SQL DONE >
```

Anche all'interno di questa struttura si possono usare le istruzioni '**`<!SQL CONTINUE>`**' e '**`<!SQL BREAK>`**'.

327.4 Istruzioni

Le istruzioni «normali» di WWW-SQL, ovvero quelle che non servono a descrivere delle strutture di controllo, sono descritte in questa sezione e in quelle seguenti. In particolare si può notare che WWW-SQL offre delle istruzioni per la lettura semplificata dell'esito di un'interrogazione SQL e altre per la lettura dettagliata, fino ad arrivare a distinguere riga per riga e campo per campo.

> È importante chiarire che, anche se un'«interrogazione» serve principalmente per leggere dati da una relazione di una base di dati, nello stesso modo, attraverso WWW-SQL si potrebbero fare delle registrazioni.

Segue un elenco di istruzioni di tipo vario, mentre nelle sezioni seguenti vengono raccolte altre istruzioni più specifiche.

- Emissione di una stringa con espansione di variabili:

  ```
  <! SQL PRINT stringa >
  ```

  L'istruzione '**`<!SQL PRINT …>`**' permette di emettere una stringa. Dal momento che un file HTML non ha bisogno di accorgimenti particolari per mostrare una stringa costante, è evidente che il senso di questa istruzione sta nella possibilità di indicare delle variabili da espandere, come nell'esempio seguente:

  ```
  <P>Il contatore ha raggiunto il livello <! SQL PRINT "$contatore" >.</P>
  ```

- Risultato di un'espressione:

  ```
  <! SQL EVAL espressione >
  ```

[3] Se l'istruzione '**`<!SQL WHILE…>`**' non viene riconosciuta, significa che non è disponibile la scansione iterativa.

L'istruzione '**<!SQL EVAL** …**>**' è simile a '**<!SQL PRINT** …**>**', con la differenza che l'argomento non è più una stringa, ma un'espressione differente, il cui risultato viene emesso alla fine.

- Impostazione di una variabile:

```
<! SQL SET nome_variabile valore_da_assegnare >
```

L'istruzione '**<!SQL SET** …**>**' permette di definire e inizializzare una variabile. L'esempio seguente definisce la variabile '**contatore**', inizializzandola a zero:

```
<! SQL SET contatore 0 >
```

- Impostazione di una variabile attraverso un'espressione:

```
<! SQL SETEXPR nome_variabile espressione >
```

L'istruzione '**<!SQL SETEXPR** …**>**' permette di definire e inizializzare una variabile; in particolare, il valore che si assegna può essere il risultato della valutazione di un'espressione. L'esempio seguente definisce la variabile '**contatore**', inizializzandola con il risultato dell'espressione '**$contatore - 1**'. In pratica viene decrementato il contenuto della variabile '**contatore**':

```
<! SQL SETEXPR contatore $contatore - 1 >
```

- Definizione di un valore predefinito per il contenuto di una variabile:

```
<! SQL SETDEFAULT nome_variabile valore_da_assegnare >
```

L'istruzione '**<!SQL SETDEFAULT** …**>**' permette di stabilire un valore predefinito per una variabile; a differenza di '**<!SQL SET** …**>**' la variabile non viene modificata se esiste già e ha un valore. L'esempio seguente definisce la variabile '**contatore**', solo se necessario, inizializzandola con il valore 10:

```
<! SQL SETDEFAULT contatore 10 >
```

- Elenco variabili:

```
<! SQL DUMPVARS >
```

L'istruzione '**<!SQL DUMPVARS>**' emette l'elenco delle variabili esistenti, assieme al valore che contengono. Può essere usato per scopo diagnostico, quando si cerca di capire cosa succede realmente.

327.4.1 Apertura e chiusura di una connessione, e accesso a una base di dati

L'interrogazione di una base di dati deve essere preceduta dalla connessione a un servente DBMS e dalla selezione di una base di dati; inoltre, al termine delle interrogazioni, si passa normalmente alla chiusura di una connessione, in pratica secondo lo schema seguente:

```
<! SQL CONNECT … >
<! SQL DATABASE nome_della_base_di_dati >
…
…
<! SQL CLOSE >
```

In breve: '**<!SQL CONNECT** …**>**' serve a iniziare una connessione con un servente per l'accesso a una base di dati; '**<!SQL DATABASE** …**>**' serve a indicare la base di dati specifica presso il servente; '**<!SQL CLOSE>**' chiude la connessione.

- Accesso a un servente DBMS:

```
<! SQL CONNECT [host [utente [parola_d'ordine]]] >
```

L'istruzione '**<!SQL CONNECT** ...**>**' permette di iniziare una connessione con un DBMS. Dipende dal DBMS stesso se è possibile accedere senza alcun sistema di autenticazione. In generale, se non si indica il nodo a cui accedere, si intende *localhost*; inoltre, se non si indica l'utente, si fa riferimento al numero UID con il quale funziona il programma servente del servizio HTTP (che a sua volta avvia il programma CGI). Se si utilizza una distribuzione GNU/Linux Debian, occorre considerare che il servente HTTP funziona con i privilegi dell'utente '**www-data**' e che si tratta di un nome impossibile per PostgreSQL. In questo caso, occorre indicare dettagliatamente tutti gli argomenti, con l'eccezione della parola d'ordine che può essere omessa se PostgreSQL è stato configurato in modo da non richiederla. L'esempio che segue richiede di connettersi al servente DBMS PostgreSQL che opera nello stesso elaboratore locale, utilizzando l'identità dell'utente '**nobody**' e senza specificare alcuna parola d'ordine:

```
<! SQL CONNECT localhost nobody >
```

- Selezione di una base di dati specifica:

```
<! SQL DATABASE nome_base_di_dati >
```

L'istruzione '**<!SQL DATABASE** ...**>**' permette di aprire una base di dati specifica; per la precisione, utilizzando PostgreSQL, l'accesso al servente avviene solo dopo che è stata specificata la base di dati.

- Chiusura di una connessione:

```
<! SQL CLOSE >
```

La chiusura di una connessione (e quindi anche di una base di dati aperta), si ottiene con l'istruzione '**<!SQL CLOSE>**'.

Prima di passare alla descrizione delle istruzioni che permettono l'interrogazione del contenuto di una base di dati, viene mostrato un esempio che si limita a elencare la tabella '**Indirizzi**' della base di dati '**anagrafe**':

```
<HTML>
<HEAD>
    <TITLE>Esempio di interrogazione</TITLE>
</HEAD>
<BODY>
<H1>Esempio di interrogazione</H1>
<! SQL CONNECT localhost nobody >
<! SQL DATABASE anagrafe >
<! SQL QUERY "SELECT * FROM Indirizzi" RICHIESTA_1 >
<! SQL QTABLE RICHIESTA_1 >
<! SQL FREE RICHIESTA_1 >
<! SQL CLOSE >
</BODY>
```

327.4.2 Istruzioni di interrogazione normali

L'interrogazione di una base di dati avviene attraverso la definizione di un riferimento, che si apre e si chiude come se fosse un flusso di file nei linguaggi di programmazione comuni. Per aprire questo riferimento si inizia con l'invio di un'interrogazione SQL; successivamente si potrà leggere l'esito dell'interrogazione attraverso il riferimento che è stato aperto; infine si passa alla chiusura del riferimento:

```
<! SQL QUERY stringa_di_interrogazione_sql riferimento >
...
...
<! SQL FREE riferimento >
```

- Apertura di un'interrogazione:

```
<! SQL QUERY stringa_di_interrogazione_sql riferimento >
```

L'istruzione '**<!SQL QUERY** ...**>**' definisce una stringa di interrogazione da inviare al servente DBMS. A questa interrogazione viene abbinato un riferimento costituito da un nome, che in seguito deve essere usato per leggere l'esito dell'interrogazione. Nell'esempio che appare nella sezione precedente, si vedeva l'istruzione seguente con la quale si selezionano tutte le righe della tabella '**Indirizzi**', abbinando questo risultato al nome '**RICHIESTA_1**':

```
<! SQL QUERY "SELECT * FROM Indirizzi" RICHIESTA_1 >
```

- Tabella rapida:

```
<! SQL QTABLE riferimento [borders] >
```

L'istruzione '**<!SQL QTABLE** ...**>**' consente di rappresentare rapidamente il risultato di un'interrogazione attraverso una tabella HTML. In particolare, utilizzando la parola chiave '**borders**', la tabella che si genera avrà i bordi delle caselle visibili. L'esempio seguente mostra in che modo visualizzare rapidamente il risultato dell'interrogazione abbinata al nome '**RICHIESTA_1**':

```
<! SQL QTABLE RICHIESTA_1 >
```

- Elenco rapido:

```
<! SQL QLONGFORM riferimento >
```

L'istruzione '**<!SQL QLONGFORM** ...**>**' si utilizza in modo simile a '**<!SQL QTABLE** ...**>**', per rappresentare il risultato di un'interrogazione attraverso un elenco dettagliato, senza una tabella HTML.

- Chiusura del riferimento all'interrogazione:

```
<! SQL FREE riferimento >
```

Come è stato mostrato all'inizio, l'istruzione '**<!SQL FREE** ...**>**' serve a chiudere il riferimento a un'interrogazione.

- Realizzazione di un elenco di voci da selezionare:

```
<! SQL QSELECT riferimento variabile_modulo_html >
```

Con l'istruzione '**<!SQL QSELECT** ...**>**' si ottiene un elenco di voci di un modulo di selezione. In generale, la cosa corrisponde a:

```
<SELECT NAME="variabile_modulo_html ">
<! SQL PRINT_ROWS riferimento "<OPTION NAME=\"@riferimento .0\">riferimento .1">
</SELECT>
```

L'istruzione '**<!SQL PRINT_ROWS** ...**>**' è descritta nella prossima sezione.

327.4.3 Istruzioni per la selezione dettagliata di righe e campi

È possibile selezionare in maniera più precisa le righe e i campi di ciò che si ottiene da un'interrogazione SQL. Attraverso l'istruzione '**<!SQL FETCH** *riferimento* **>**' si preleva la riga attuale dall'interrogazione a cui si fa riferimento. Questo prelievo permette di fare riferimento agli elementi della riga attraverso una notazione particolare:

```
@riferimento .n
```

In pratica, è come se fosse l'espansione di una variabile, con la differenza che si indica il nome di un riferimento a un'interrogazione aperta, aggiungendo un'estensione numerica, separata da un punto, dove lo zero corrisponde al primo campo e n - 1 corrisponde al campo n-esimo.

- Modifica della riga attuale all'interno del risultato di un'interrogazione:

```
<! SQL SEEK riferimento n_riga >
```

L'istruzione '**<!SQL SEEK** ...**>**' permette di modificare la riga attuale all'interno di un'interrogazione. Per indicare il numero della riga, occorre tenere presente che lo zero corrisponde alla prima. L'esempio seguente fa in modo che la riga attuale diventi la seconda del riferimento '**RICHIESTA_1**':

```
<! SQL SEEK RICHIESTA_1 1 >
```

- Prelievo della riga attuale di un certo riferimento:

```
<! SQL FETCH riferimento >
```

L'istruzione '**<!SQL FETCH** ...**>**' permette di rendere disponibile il contenuto della riga attuale di un certo riferimento. L'esempio seguente preleva il contenuto della riga attuale del riferimento '**RICHIESTA_1**'; quindi mostra il primo e il secondo campo di questa riga, che si presume corrispondano al cognome e al nome di una persona:

```
<! SQL FETCH RICHIESTA_1 >
<P>Cognome: <! SQL PRINT "@RICHIESTA_1.0" ></P>
<P>Nome: <! SQL PRINT "@RICHIESTA_1.1" ></P>
```

- Emissione di una stringa per ogni riga:

```
<! SQL PRINT_ROWS riferimento stringa >
```

L'istruzione '**<!SQL PRINT_ROWS** ...**>**' è una sorta di istruzione '**<!SQL PRINT** ...**>**' ripetuta per tutte le righe di un'interrogazione, a partire da quella corrente. L'esempio seguente mostra la visualizzazione dei primi due campi di tutte le righe di un'interrogazione, a cui si fa riferimento con il nome '**Q**':

```
<! SQL SEEK Q 0 >
<! SQL PRINT_ROWS Q "<P>Cognome: @Q.0</P>\n<P>Nome: @Q.1</P>\n" >
```

L'esempio seguente mostra la realizzazione di un modulo per la selezione di un articolo, attraverso l'invio del codice corrispondente. A questo proposito, si suppone che la prima colonna del risultato dell'interrogazione a cui si fa riferimento con il nome '**ELENCO**', corrisponda al codice dell'articolo, mentre la seconda corrisponda a una sua descrizione:

```
<p><FORM ACTION=ordine.pgsql>
<SELECT NAME="codice">
<! SQL PRINT_ROWS ELENCO "<OPTION NAME=\"@ELENCO.0\">@ELENCO.1" >
</SELECT>
<INPUT TYPE="submit">
</FORM></p>
```

Dal momento che si fa riferimento alle prime due colonne, la stessa cosa avrebbe potuto essere realizzata con l'istruzione '**<!SQL QSELECT** ...**>**', nel modo seguente:

```
<p><FORM ACTION=ordine.pgsql>
<! SQL QSELECT ELENCO codice >
<INPUT TYPE="submit">
</FORM></p>
```

327.5 Riferimenti

- James Henstridge, *WWW-SQL*

  <http://www.daa.com.au/~james/www-sql/www-sql.html>

Le funzioni e i trigger in PostgreSQL: un'esercitazione didattica

Questo capitolo deriva da un lavoro originale di Antonio Bernardi <*mailto:brngb@tin.it*> che ne ha concesso espressamente l'utilizzo libero all'interno di questa opera. Il testo originale di questo lavoro si trova presso <*http://linuxdidattica.org/docs/besta/*>.

328.1 Lo scenario: la base di dati «biblioteca»

L'esercitazione che viene proposta, riguarda la gestione dei prestiti di una biblioteca. Le tabelle coinvolte sono:

- '**libri(n_inv,autore,titolo,collocazione,soggetto,cod_ed,prezzo,anno_ed)**'
- '**utenti(cod_ut,nome,cognome,telefono,indirizzo,citta)**'
- '**localita(citta,cap,prov,naz)**'
- '**editori(cod_ed,rag_soc,indirizzo,citta,telefono)**'
- '**prestiti(np,n_inv,cod_ut,data_p,data_r)**'

Le tabelle sono ovviamente ridotte all'osso e sono autoesplicative. In questo esercizio la tabella fondamentale è la tabella prestiti, nella quale troviamo il codice dell'utente ('**cod_ut**') e il numero di inventario del libro ('**n_inv**'). Quando viene fatto un prestito l'impiegato dovrà inserire:

- il numero di inventario del libro, '**n_inv**';
- il codice dell'utente, '**cod_ut**';
- la data del prestito, '**data_p**'.

A questo punto, per evitare errori e non immettere un libro che è già a prestito, si dovrà fare un controllo che impedisca tale inserimento. Nella tabella prestiti vi è il campo '**data_r**' che riporta la data di rientro di un libro. Quando un libro viene dato a prestito questa data verrà inserita in modo predefinito usando un valore assurdo (2050.01.01) che serve a indicare che il libro è a prestito. Conseguentemente, quando si inserisce un prestito nuovo, se nella tabella prestiti esiste una riga con un attributo '**n_inv**' uguale a quello che si vuole inserire e '**data_r**' equivalente al valore convenzionale indicante che il libro è a prestito, il DBMS deve avvisare dell'errore.

Nell'esercitazione seguente questo controllo viene fatto in due modi differenti: prima con l'utilizzo di una funzione, poi con l'utilizzo di un trigger.

328.2 In classe: al lavoro!

Si immagina di avere un'unico elaboratore, nel quale sia già installato PostgreSQL.

Per prima cosa ci si deve collegare all'elaboratore GNU/Linux, come utente '**postgres**', per creare la base di dati:

```
postgres$ createdb biblioteca
```

Successivamente ci si connette alla base di dati '**biblioteca**' con il programma cliente '**psql**' per creare le tabelle:

```
postgres$ psql -h localhost -d biblioteca -U postgres[ Invio ]

biblioteca=> create table localita ([ Invio ]

biblioteca-> citta char(20) primary key,[ Invio ]

biblioteca-> cap char(5),[ Invio ]

biblioteca-> prov char(2),[ Invio ]

biblioteca-> naz char(3)[ Invio ]

biblioteca-> );[ Invio ]

biblioteca=> create table editori ([ Invio ]

biblioteca-> cod_ut char(3) primary key,[ Invio ]

biblioteca-> nome char(10),[ Invio ]

biblioteca-> cognome char(10),[ Invio ]

biblioteca-> telefono char(11),[ Invio ]

biblioteca-> indirizzo char(20),[ Invio ]

biblioteca-> citta char(20) references localita[ Invio ]

biblioteca-> );[ Invio ]

biblioteca=> create table libri ([ Invio ]

biblioteca-> n_inv char(5) primary key,[ Invio ]

biblioteca-> autore char(25),[ Invio ]

biblioteca-> titolo char(35) not null,[ Invio ]

biblioteca-> collocazione char(15),[ Invio ]

biblioteca-> soggetto char(11),[ Invio ]

biblioteca-> cod_ed char(5) references editori,[ Invio ]

biblioteca-> prezzo integer,[ Invio ]

biblioteca-> anno_ed char(4)[ Invio ]

biblioteca-> );[ Invio ]

biblioteca=> create table utenti ([ Invio ]
```

```
biblioteca-> cod_ut char(5) primary key,[ Invio ]

biblioteca-> nome char(10),[ Invio ]

biblioteca-> cognome char(10),[ Invio ]

biblioteca-> telefono char(11),[ Invio ]

biblioteca-> indirizzo char(20),[ Invio ]

biblioteca-> citta char(20) references localita[ Invio ]

biblioteca-> );[ Invio ]

biblioteca=> create table prestiti ([ Invio ]

biblioteca-> np serial,[ Invio ]

biblioteca-> n_inv char(5) references libri,[ Invio ]

biblioteca-> cod_ut char(5) references utenti,[ Invio ]

biblioteca-> data_p date check (data_p <= data_r),[ Invio ]

biblioteca-> data_r date default '2050.1.1'[ Invio ]

biblioteca-> );[ Invio ]
```

A questo punto si vanno a popolare le tabelle (esclusa la tabella '**prestiti**'): qui bisogna fare attenzione, inserendo prima i dati delle tabelle che non hanno chiavi esterne; successivamente inserendo quelle tabelle che fanno riferimento alle prime tramite chiavi esterne. Per esempio, è necessario popolare la tabella '**localita**' prima della tabella '**editori**'.

```
biblioteca=> insert into localita (citta, cap, prov, naz)[ Invio ]

biblioteca-> values ('TREVISO', '31100', 'TV', 'I');[ Invio ]

biblioteca=> insert into localita (citta, cap, prov, naz)[ Invio ]

biblioteca-> values ('PADOVA', '35100', 'PD', 'I');[ Invio ]

biblioteca=> insert into localita (citta, cap, prov, naz)[ Invio ]

biblioteca-> values ('MILANO', '20100', 'MI', 'I');[ Invio ]

biblioteca=> insert into editori (cod_ed, rag_soc, indirizzo, ↩
↪citta, telefono)[ Invio ]

biblioteca-> values ('1', 'CEDAM SPA', 'VIA JAPPELLI 5/6', ↩
↪'PADOVA', '049-8239111');[ Invio ]

biblioteca=> insert into editori (cod_ed, rag_soc, indirizzo, ↩
↪citta, telefono)[ Invio ]

biblioteca-> values ('2', 'ELEMOND SPA', 'VIA ROMA 17', ↩
↪'MILANO', '02-7820012');[ Invio ]
```

```
biblioteca=> insert into utenti (cod_ut, nome, cognome, ↩
↪telefono, indirizzo, citta)[ Invio ]

biblioteca-> values ('1', 'LUCA', 'BONALDO', '0422-401582', ↩
↪'VIA CORNARE 14', 'TREVISO');[ Invio ]

biblioteca=> insert into utenti (cod_ut, nome, cognome, ↩
↪telefono, indirizzo, citta)[ Invio ]

biblioteca-> values ('2', 'LUIGI', 'GOBBO', '049-458270', ↩
↪'VIA MANIN 72', 'PADOVA');[ Invio ]

biblioteca=> insert into utenti (cod_ut, nome, cognome, ↩
↪telefono, indirizzo, citta)[ Invio ]

biblioteca-> values ('3', 'SIMONE', 'PRIAMO', '0422-478791', ↩
↪'VIALE M.GRAPPA 1', 'TREVISO');[ Invio ]

biblioteca=> insert into utenti (cod_ut, nome, cognome, ↩
↪telefono, indirizzo, citta)[ Invio ]

biblioteca-> values ('4', 'MAURO', 'MENEGAZZI', '049-987756', ↩
↪'VIA EVEREST 7', 'PADOVA');[ Invio ]

biblioteca=> insert into libri (n_inv, autore, titolo, ↩
↪collocazione, soggetto, cod_ed, prezzo, anno_ed)[ Invio ]

biblioteca-> values ('1', 'STELLIO MARTELLI', ↩
↪'RACCONTI MITOLOGICI', 'X.1.1', 'STORICO', '1', 7000, '1992');[ Invio ]

biblioteca=> insert into libri (n_inv, autore, titolo, ↩
↪collocazione, soggetto, cod_ed, prezzo, anno_ed)[ Invio ]

biblioteca-> values ('2', 'HECTOR MALOT', 'SENZA FAMIGLIA', ↩
↪'X.1.2', 'DRAMMATICO', '2', 14000, '1990');[ Invio ]

biblioteca=> insert into libri (n_inv, autore, titolo, ↩
↪collocazione, soggetto, cod_ed, prezzo, anno_ed)[ Invio ]

biblioteca-> values ('3', 'LOUISE MAY ALCOTT', ↩
↪'PICCOLE DONNE CRESCONO', 'X.1.3', 'ROMANTICO', '1', 10000,
'1991');[ Invio ]

biblioteca=> insert into libri (n_inv, autore, titolo, ↩
↪collocazione, soggetto, cod_ed, prezzo, anno_ed)[ Invio ]

biblioteca-> values ('4', 'MARY E. MAPES DODGE', ↩
↪'PATTINI D ARGENTO', 'X.1.4', 'FANTASTICO', '2', 13000, '1987');[ Invio ]
```

A questo punto se si inseriscono i dati nella tabella '**prestiti**' ci si può trovare nella situazione di avere a prestito lo stesso libro più volte contemporaneamente. Naturalmente si riesce a sperimentare facilmente tale situazione con qualche prova.

Per esempio, se viene digitato

```
biblioteca=> insert into prestiti (n_inv, cod_ut, data_p)[ Invio ]
```

```
biblioteca-> values ('2','3', '2001.1.1');[ Invio ]
```

si inserisce una riga sulla tabella prestiti. Se si digita ancora

```
biblioteca=> insert into prestiti (n_inv, cod_ut, data_p)[ Invio ]

biblioteca-> values ('2','1', '2001.1.1');[ Invio ]
```

ci si trova con un libro che è dato a prestito all'utente di codice '**3**' e contemporaneamente all'utente di codice '**1**', il che è assurdo, assumendo il fatto che un libro non possa essere preso a prestito da più utenti, contemporaneamente.

La soluzione proposta utilizza le funzioni e i trigger di PostgreSQL. Per la realizzazione delle funzioni si mostra qui l'uso del linguaggio Plpgsql, che prima di poter essere utilizzato deve essere associato esplicitamente. Questa operazione richiede l'uso del comando '**createlang**', come si vede nell'esempio seguente:

```
postgres$ createlang plpgsql -h localhost -d biblioteca ↩
↪--pglib=/usr/lib/pgsql
```

In questo caso, si intende che il file '`plpgsql.so`' sia contenuto nella directory '`/usr/lib/pgsql/`'.

328.2.1 Le funzioni

Successivamente si passa alla scrittura della funzione che viene mostrata sotto, con l'aiuto di un programma per la creazione e la modifica di file di testo (come VI per esempio), generando il file '`funzione_controlla.plpgsql`'.[1]

```
create function inserisci_prestito(char(5), char(5), date)
returns boolean
     as 'declare
         numero_inventario alias for $1;
         codice_utente alias for $2;
         data_prestito alias for $3;
         data_restituzione date;
         prestito record;
         begin
            data_restituzione:=''2050.1.1'';
            select into prestito *
            from prestiti
            where n_inv=numero_inventario and
            data_r=data_restituzione;
            if found
               then
                  raise exception \'il libro è già a prestito\';
                  return ''f'';
               else
                  insert into prestiti (n_inv, cod_ut, data_p)
                  values (numero_inventario, codice_utente, data_prestito);
                  return ''t'';
            end if;
         end;'
language 'plpgsql';
```

Inizialmente si assegnano alle variabili '**numero_inventario**', '**codice_utente**' e '**data_prestito**' i valori corrispondenti '**n_inv**', '**cod_ut**' e '**data_p**'. Successivamente viene definita la variabile '**data_restituzione**', di tipo '**date**', alla quale viene assegnato il valo-

[1] Volendo rimanere nell'ambito di '**psql**', si può usare il comando '**\!**' per avviare temporaneamente il programma di creazione e modifica dei file di testo.

re sentinella '**2050.1.1**'; quindi la variabile '**prestito**', di tipo '**record**', che dovrà contenere la riga letta dalla tabella '**prestiti**', nel caso la lettura vada a buon fine con l'istruzione:

```
select into prestito * from prestiti where...
```

Sostanzialmente si legge la tabella prestiti e se si trova una riga che soddisfa la condizione di uguaglianza tra '**n_inv**' e il numero di inventario del libro che si vuole dare a prestito e tra la '**data_r**' e la data fittizia del 2050.01.01, significa che il libro è già a prestito.

Se questa riga viene trovata (con la condizione '**if found**'), la funzione deve uscire dal blocco begin-end ed emettere un avviso che il libro è già a prestito, altrimenti deve inserire la riga in oggetto nella tabella prestiti con l'istruzione

```
insert into prestiti ...
```

con i dati passati dalla funzione.

Una volta scritto il file della funzione, si deve acquisirne il codice con il comando seguente, nell'ambito di '**psql**':

```
biblioteca=> \i funzione_controlla.plpgsql[ Invio ]
```

Se nel frattempo la tabella '**prestiti**' contiene righe senza senso, conviene azzerarla completamente, prima di mettere in pratica l'uso della nuova funzione di controllo:

```
biblioteca=> \delete from prestiti;[ Invio ]
```

Per l'uso vero e proprio della funzione, si interviene come nell'esempio seguente:

```
biblioteca=> select inserisci_prestito ('2','3', ↩
↪cast '2001.1.1' as date);[ Invio ]
```

A questo punto, se si tenta di inserire per due volte lo stesso prestito, la funzione impedisce l'operazione e avvisa dell'errore.

328.2.2 I trigger e le funzioni

L'utilizzo dell'istruzione '**select**' abbinata a una funzione può creare qualche confusione. Si può superare questo problema utilizzando un trigger che richiami automaticamente una funzione di controllo. Quello che segue è l'esempio di tale funzione corrispondente al file '`funzione_trigger.plpgsql`'.

```
create function inserisci_prestito_trigger()
returns opaque
   as 'declare
     numero_inventario char(5);
     data_restituzione date;
     prestito record;
        begin
        numero_inventario:=new.n_inv;
        data_restituzione:=''2050.1.1'';
           select into prestito *
           from prestiti
           where n_inv=numero_inventario and
           data_r=data_restituzione;
              if found
                 then
                    raise exception \'il libro è già a prestito\';
                 else
                    return new;
               end if;
        end;'
```

```
language 'plpgsql';

create trigger controlla_libro_uscito
before insert
on prestiti
for each row
execute procedure inserisci_prestito_trigger();
```

Nel file in questione, si vede la dichiarazione di una funzione analoga a quanto già mostrato in precedenza, seguita dalla dichiarazione del trigger relativo.

La variabile new corrisponde alla nuova riga che si vuole inserire con l'istruzione '**insert into**', ed è di tipo '**record**'.

Dopo averle dichiarate, si assegna alla variabile '**numero_inventario**' il valore '**new.n_inv**' e alla variabile '**data_restituzione**' il valore sentinella '**2050.1.1**'. Successivamente con l'istruzione

```
select into prestito * ...
```

si va a vedere se nella tabella '**prestiti**' esiste una riga che soddisfa la condizione di esistenza del libro a prestito. Se si trova questa riga viene mostrato un messaggio di errore, altrimenti la funzione deve restituire il valore contenuto nella variabile '**new**', ovvero la riga che verrà inserita nella tabella.

La funzione '**inserisci_prestito_trigger()**' viene messa in azione, ogni volta che si vuole inserire una riga nel file prestiti, attraverso il controllo del trigger '**controlla_libro_uscito**'.

Si acquisisce la funzione e il trigger con il comando seguente, nell'ambito di '**psql**':

```
biblioteca=> \i funzione_trigger.plpgsql[ Invio ]
```

A questo punto per inserire un libro a prestito si userà l'istruzione standard:

```
biblioteca=> insert into prestiti (n_inv, cod_ut, data_p) ↩
↪values ('3','2','2001.10.10');[ Invio ]
```

Se il libro non è già a prestito, si ottiene la segnalazione standard del fatto che il libro è stato inserito. Se si ritenta l'inserimento di un prestito con lo stesso numero di inventario, si ottiene solo la segnalazione di errore prevista.

```
biblioteca=> insert into prestiti (n_inv, cod_ut, data_p) ↩
↪values ('3','3','2001.10.10');[ Invio ]

ERROR: il libro è già a prestito
```

328.3 Riferimenti

- Bruce Momjian, *PostgreSQL, Introduction and concepts*, capitolo *Functions and trigger*

  <http://www.postgresql.org/docs/aw_pgsql_book/index.html>

- The PostgreSQL Global Development Group, *PostgreSQL, Programmer's guide*, capitolo *Pl/pgSQL SQL procedural language*

  <http://www.postgresql.org/idocs/index.php?programmer.html>

Indice analitico del volume

`/etc/postgresql/pg_hba.conf`, 3677
`/etc/postgresql/postmaster.init`, 3677
AWK, 3592, 3611
basi di dati, 3631
BRE, 3572
CGI, 3714
CGI: accesso a una base di dati, 3714
`createdb`, 3679
`createuser`, 3673
DBA, 3632
DBMS, 3631
DBMS: DBA, 3632
DBMS: DDL, 3632
DBMS: DML, 3632
DDL, 3632
`destroyuser`, 3673
DML, 3632
`dropdb`, 3679
ERE, 3572
espressione regolare, 3572, 3580
`initdb`, 3668
ISO 8601, 3645
M4, 3618
M4: `changecom`, 3626
M4: `define`, 3622
M4: `divert`, 3627
M4: `dnl`, 3626
M4: `forloop`, 3625
M4: `ifdef`, 3624
M4: `ifelse`, 3624
M4: `include`, 3626
M4: `shift`, 3625
M4: `sinclude`, 3626
M4: `undefine`, 3624
M4: `undivert`, 3627
`PAGER`, 3684
PgAccess, 3706
`pg_dump`, 3688, 3690
`pg_passwd`, 3676
PostgreSQL, 3667, 3706
PostgreSQL: autenticazione, 3675
PostgreSQL: LibPQ, 3680
`postmaster`, 3670
programmazione: PostgreSQL, 3667
programmazione: SQL, 3642, 3692
programmazione: WWW-SQL, 3714
`psql`, 3680, 3681
RDBMS, 3632
RE, 3572
regexp, 3572, 3580

relazione, 3632
SED, 3584
SQL, 3642, 3692, 3714
SRE, 3572
trigger, 3725
tupla, 3632
~postgres/pg_hba.conf, 3675
~postgres/pg_shadow, 3673
~postgres/pg_user, 3673
$PGDATA, 3667, 3668
$PGHOST, 3680
$PGLIB, 3668
$PGPORT, 3680